Data Distribution for Fog-Based IoT Applications

vorgelegt von
M.Sc.
Jonathan Hasenburg
ORCID: 0000-0001-8549-0405

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Florian Tschorsch
Gutachter: Prof. Dr. David Bermbach
Gutachter: Prof. Dr. Jochen Schiller
Gutachter: Prof. Dr. Odej Kao

Tag der wissenschaftlichen Aussprache: 25. Juni 2021

Berlin 2021

Abstract

Fog computing is an emerging computing paradigm that employs edge devices, machines within the core network, and the cloud. Distributing IoT data via fog-based pub/sub systems has many advantages, including low latency communication between physically close clients, better availability, and reduced bandwidth consumption in the wide-area network. For building, testing, and evaluating such fog-based pub/sub systems, we identified two main needs: First, the need for considering the unique characteristics of the fog and the IoT. Second, the need for automated execution of fog application experiments in a controllable environment. In this thesis, we present three main contributions to address these needs:

Our first contribution is BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems. BCGroups can be used with existing cloud-based pub/sub offers. For routing messages between brokers fast and efficiently, BCGroups uses IoT-specific domain knowledge: low communication latency is often only required between devices in physical proximity.

Our second contribution comprises GeoBroker & DisGB, two pub/sub broker systems that leverage geo-context for IoT data distribution. In many IoT scenarios, geo-context information is readily available, i.e., publishers often know where their data is relevant, and subscribers can often specify where relevant data originates. When running on a single machine, GeoBroker reduces the load on the broker system, bandwidth consumption, and the number of irrelevant messages that subscribers need to process. DisGB builds upon GeoBroker and also comprises two novel strategies for inter-broker message routing in multi-machine setups. Both strategies achieve a similar communication latency as flooding events or subscriptions while requiring significantly less inter-broker messages.

Our third contribution is MockFog, an approach for the automated execution of fog application experiments in the cloud. The main idea is to use an emulated infrastructure testbed that can be manipulated based on a predefined orchestration schedule. This way, fog applications and fog systems can run in the cloud while experiencing comparable performance and failure characteristics as in a real fog deployment. Moreover, it allows application engineers to test arbitrary failure scenarios and various infrastructure options at large scale.

Kurzdarstellung

Fog Computing ist ein gerade entstehendes Rechenparadigma, welches die gleichzeitige Nutzung von Edge-Ressourcen, von Maschinen innerhalb des Kernnetzwerkes und der Cloud ermöglicht. Die Nutzung von Fog-Ressourcen zur Verteilung von IoT-Daten mittels Pub/Sub-Systemen hat viele Vorteile; dazu gehört unter anderem die Möglichkeit für physisch nahe Geräte mit geringer Latenz miteinander zu kommunizieren, eine erhöhte Systemverfügbarkeit und eine geringe Nutzung von Bandbreitenressourcen in überregionalen Netzwerken. Um solche Fog-basierten Pub/Sub-Systeme zu entwickeln, zu testen und zu evaluieren, wurden zwei grundlegende Erfordernisse identifiziert: Erstens die Notwendigkeit IoT-spezifische und Fog-spezifische Charakteristiken zu berücksichtigen. Zweitens der Bedarf an einer automatisierten Durchführung von Experimenten mit Fog-Anwendungen in einer kontrollierbaren Umgebung. Zur Erfüllung dieser Erfordernisse leistet diese Dissertation drei entscheidende Beiträge:

Der erste Beitrag ist BCGroups, eine Strategie zur Verteilung von IoT-Daten innerhalb von Pub/Sub-Systemen in Fog-Umgebungen. BCGroups ermöglicht dabei auch die Einbindung bereits bestehender Pub/Sub-Angebote. Für die Verteilung von Nachrichten zwischen einzelnen Brokern wird IoT-spezifisches Domänenwissen genutzt: Nachrichtenaustausch mit geringer Latenz wird überwiegend von Geräten, die sich in physischer Nähe zueinander befinden, benötigt.

Der zweite Beitrag beinhaltet GeoBroker & DisGB, zwei Pub/Sub-Systeme, die das Verteilen von IoT-Daten mittels Geo-Kontexten optimieren. Geo-Kontextinformationen sind in vielen IoT-Szenarien vorhanden, so wissen Publisher häufig, wo ihre Daten von Relevanz sind, während Subscriber häufig spezifizieren können, woher relevante Daten kommen. Durch die Nutzung dieser Informationen reduziert GeoBroker in zentralisierten Setups die Systemlast, die Nutzung von Bandbreitenressourcen, sowie die Anzahl an irrelevanten Nachrichten, die von Subscribern verarbeitet werden müssen. DisGB baut auf GeoBroker auf und beinhaltet zusätzlich zwei neue Strategien zur Verteilung von Nachrichten zwischen Brokern. Beide DisGB-Strategien erreichen eine vergleichbare Latenz beim Nachrichtenaustausch wie das direkte Verteilen von Events oder Subscriptions an alle Broker, aber sie resultieren in einem deutlich geringerem Nachrichtenaufkommen.

Der dritte Beitrag ist MockFog, ein Ansatz zur automatisierten Ausführung von Experimenten mit Fog-Anwendungen in der Cloud. Die grundlegende Idee ist die Emulierung einer Infrastrukturtestumgebung, welche basierend auf einem vordefinierten Orchestrierungsplan manipuliert werden kann. Dadurch können Fog-Anwendungen und Fog-Systeme vollständig in der Cloud betrieben werden, während sie ähnliche Performance- und Fehlereinflüsse wie in ihrer tatsächlichen Fog-Umgebung erfahren. Zusätzlich ermöglicht dies das Testen von beliebigen Fehlerszenarien auf verschiedensten Infrastrukturoptionen im großem Maßstab.

Danksagung

Eine Dissertation wird niemals in absoluter Isolation verfasst. Ideen müssen sich entwickeln und wachsen können, was am besten durch einen regen Austausch mit anderen Wissenschaftlern gelingt. Unter anderem ist es auch aus diesem Grund besonders anspruchsvoll, ein solches Unterfangen zum Abschluss zu bringen, wenn man wegen einer Pandemie zu Hause bleiben muss, anstatt sich in der Universität sowie auf Konferenzen mit Kollegen austauschen zu können. Aus diesem Grund möchte ich zuerst dem Team des Mobile-Cloud-Computing–Lehrstuhls danken, welches mir in den täglichen Zoom-Meetings geholfen hat auf Kurs zu bleiben und einen kühlen Kopf zu bewahren.

Besonderen Dank möchte ich auch meinem Mentor, Prof. Dr. Bermbach, aussprechen. Schon während meines Masterstudiums habe ich seine methodische und gründliche Arbeitsweise kennen und schätzen gelernt, weswegen ich auch besonders erfreut war, als er mir das Angebot unterbreitete als erster wissenschaftlicher Mitarbeiter den Aufbau des neu gegründeten Lehrstuhls zu unterstützen. Zwar bedeutet es eine zusätzliche Herausforderung etwas Neues zu erschaffen, anstatt Teil eines bereits funktionierenden Teams zu werden, doch diesen Nachteil wusste Prof. Dr. Bermbach durch sein hervorragendes Mentoring mehr als auszugleichen. Unter seiner Leitung durfte ich mein Dissertationsthema nach eigenen Wünschen festlegen und gestalten, dafür bin ich ihm ebenfalls sehr dankbar.

Neben Prof. Dr. Bermbach möchte ich auch dem Vorsitzenden des Promotionsausschusses, Prof. Dr. Tschorsch, sowie den weiteren Gutachtern des Promotionsausschusses, Prof. Dr. Schiller und Prof. Dr. Kao, danken. Mit ihrem konstruktiven Feedback haben Sie mir Vertrauen in meine eigene Forschung gegeben und aufgezeigt, wie ich wichtige Beiträge noch besser und konsistenter darstellen kann.

Abschließend möchte ich mich bei all den anderen Menschen bedanken, die mich in den letzten Jahren so tatkräftig auf die verschiedensten Weisen unterstützt haben. Hervorheben möchte ich hierbei vor allem die Rolle meiner Eltern und meiner Frau. Obwohl mein Vater selbst kein Informatiker ist, hat er es trotzdem stets geschafft, sich in kürzester Zeit in meine Themen hineinzudenken und in den teilweise sehr langen und komplexen Forschungsarbeiten sogar die kleinsten Inkonsistenzen und Fehler zu finden. Vor dieser Leistung verspüre ich höchsten Respekt und Dankbarkeit. Auch möchte ich meinem Vater für seinen Glauben in mich danken: Er hat von Anfang an nie daran gezweifelt, dass ich eine Promotion anstreben und erfolgreich abschließen werde. Meiner Mutter und meiner Frau möchte ich besonders für ihre bedingungslose Liebe danken, welche mir die emotionale Stabilität verlieh, diese Dissertation zu verfassen. Bei meiner Frau möchte ich mich auch dafür bedanken, dass sie mich alltäglich in allen Belangen unterstützt hat und während der Hochs und Tiefs dieser Zeit an meiner Seite stand.

Jn.H.

Table of Contents

Acronyms

AWS Amazon Web Services

EC2 Elastic Compute Cloud

DisGB Distributed GeoBroker

EG Event Geofence

IoT Internet of Things

IoV Internet of Vehicles

LB Local Broker

LCM Leadership Capability Measure

MDL Message Delivery Latency

P2P Peer-to-Peer

Pub/Sub Publish/Subscribe

QoS Quality of Service

RP Rendezvous Point

SG Subscription Geofence

VM Virtual Machine

Part I

Foundations

Chapter 1

Introduction

The widespread deployment of connected devices in the Internet of Things (IoT) has substantially increased the amount of generated data. Today's IoT applications use this data to enable more sophisticated application scenarios. The current go-to approach for building IoT applications is transmitting device data to the cloud for processing and sending back instructions [72], e.g., to switch on a light in the presence of movement in a smart home scenario [145]. Due to its simplicity, many commercial services use this approach, e.g., AWS IoT [162] or the Azure IoT Hub [163]. However, disadvantages include long response times, unnecessary data transmissions, and the risk of exposing sensitive data to third parties [17].

The emerging fog computing paradigm promises to address these disadvantages by providing low latency communication while also keeping the scalability aspects of the cloud [17, 117]. For low latency, application components are deployed near devices or end users (also referred to as edge), e.g., as done by AWS Greengrass [161]. This can also reduce bandwidth consumption, mitigate privacy risks, and enable the edge to keep operating in the presence of network partitions. For high scalability, application components can leverage stronger machines such as cloudlets [155, 156] within the core network [17] or run directly in the cloud. This encompassing execution environment is commonly referred to as *fog* [17, 22] and comprises edge devices, machines within the core network, and the cloud.

For asynchronous, loosely coupled communication between a large number of IoT devices, applications often use broker-based publish/subscribe systems (pub/sub) [9, 102, 142]. Here, client devices create subscriptions (as subscribers) and send events (as publishers) to brokers, which then match incoming events with created subscriptions and deliver them accordingly. In a fog environment, brokers are deployed at multiple sites. Client devices connect to the nearest broker for low latency communication with other devices in physical proximity. The brokers route messages to other brokers so that devices that are not connected to the same broker can communicate. For this inter-broker routing, there are multiple strategy options; in general, they can be classified into the three categories *flooding*, *gossiping*, and *selective* [6, 154]: With flooding, brokers broadcast events or subscriptions to all other brokers. While this ensures minimum end-to-end latency, it does not scale well as every broker has to process all events or subscriptions from every other broker; this also leads to a high network load. On the other end of the spectrum, selective approaches limit the propagation of events or subscriptions to a small subset of brokers, improving system efficiency but increasing latency.

1.1 Problem Statement

For distributing IoT data across the fog, it becomes clear that there is a tradeoff between latency and excess data dissemination when looking at the two inter-broker routing strategy categories flooding and selective. Existing solutions, e.g., [2, 142], do either not address the tradeoff or require a holistic view on all events and subscriptions. Other solutions, e.g., [102, 193], might route messages across any node even though the node might not have access to sufficient compute or networking resources. Particularly in the fog, this is not acceptable due to the heterogeneity of machines, as well as partly unstable and relatively slow network connections.

Furthermore, a unique characteristic of IoT applications is that data generated by IoT devices is often only relevant to physically close devices or devices in a specific physical area. Today's pub/sub systems do not use such IoT-specific domain knowledge when deciding what data to distribute to which clients. Hence, they disregard information that is readily available in many IoT scenarios which can help to improve the delivery precision of clients' events and messages between brokers. For example, a car that aims to avoid traffic jams needs to process only data from roadside equipment and cars within its current surroundings to determine an optimal route and velocity. Therefore, from the subscribers' perspective, data originating outside an area of interest is not relevant and can be discarded. A publisher, on the other hand, might already know that provided data is only relevant for subscribers in a specific area and thus aim to prevent others from receiving it. E.g., only drivers in the immediate vicinity of a particular car might need to know its acceleration and deceleration profile, which also prevents data misuse. Such scenarios are the reason why Bellavista et al. [15] argue that geographical co-location should be considered. Other domains with applications in which the value of information depends on the location of data producers and consumers include the Internet-of-Vehicle [56, 158], Smart Cities [150], or Mobile Health [123].

Finally, evaluating fog applications and fog systems is difficult. While basic design questions can be decided using simulation, e.g., [25, 69, 84], there comes a point that requires running experiments. Due to a highly distributed execution environment, however, setting up physical testing infrastructure and managing application components is more complex than the ease of adoption developers are used to from the cloud [17]. This also makes it hard to achieve a high level of reproducibility and controllability; achieving both, however, is a main requirement for real-time system experiments [1, p. 263].

1.2 Contributions

To address the problems outlined above, we identified two main needs: First, the need for efficient pub/sub systems that take into account the unique characteristics of the fog and the IoT. Second, the need for automated execution of fog application experiments in a controllable environment. To address these needs, we present three main contributions as part of this thesis:

1.2.1 BCGroups: An Inter-Broker Routing Strategy for the Fog

We propose BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems. BCGroups builds upon IoT-specific domain knowledge: low communication latency is often only required between devices in physical proximity [56, 123, 150, 158]. Thus, the main idea is to split the set of fog brokers into well connected *broadcast groups* which use flooding for intra-group communication and a cloud relay for inter-group communication. This minimizes the communication latency of IoT devices in physical proximity. Flooding also handles frequently updated subscriptions of mobile clients particularly well as messages are sent preemptively to all brokers at which a client could create its subscriptions. Devices that are located further away can usually tolerate higher communication latency. At the same time, however, one needs to minimize excess data dissemination to not overuse precious bandwidth resources. Thus, each broadcast group elects a leader, which communicates on behalf of the group with the cloud broker and through it with other group leaders. With the broadcast group size, we have a tunable parameter for managing the tradeoff between excess data and latency. We show the effectiveness of BCGroups and that involved overheads remain manageable—this even applies to global deployments with thousands of individual broker instances. Furthermore, we highlight that BCGroups can be used in conjunction with existing cloud-based pub/sub offers.

We have published this contribution in [82].

1.2.2 GeoBroker & DisGB: Leveraging Geo-Context for IoT Data Distribution

We propose to leverage geo-context for IoT data distribution within the fog. For this purpose, we derive a generally applicable geo-context definition that comprises four dimensions: publisher location, subscriber location, event geofence, and subscription geofence. While this kind of domain knowledge is not available in every IoT scenario, using it significantly improves data distribution efficiency.

As part of this contribution, we present two pub/sub broker systems: We built GeoBroker to run on a single machine. GeoBroker uses an efficient subscription indexing structure that also stores geo-context information and an extended event/subscription matching process: Besides the normal *ContentCheck*, e.g., does the topic of a published event match the topic of a subscription, our matching process has two additional *GeoChecks*. With the event GeoCheck, publishers can control which subscribers receive messages based on the event geofence and subscriber location. With the subscription GeoCheck, subscribers can control from which publishers they receive messages based on the subscription geofence and publisher location. GeoBroker reduces the number of transmitted messages for scenarios where geo-context matters, thus reducing the broker system load, bandwidth consumption, and the number of messages that need to be processed by subscribers. While this comes with the cost of additional computation effort for the broker, we show that this overhead is relatively small at low load levels and is, at higher load levels, more than offset by the performance improvements gained by only transmitting relevant messages.

DisGB (distributed GeoBroker) builds upon GeoBroker and also uses two novel strategies for inter-broker message routing in multi-machine setups. DisGB brokers can use geo-context information to select rendezvous points (RPs) close to the publishers or subscribers of an event. This reduces the event delivery latency by up to 22 times compared to the only state-of-the-art alternative that sends slightly fewer messages. In addition, this results in significantly less inter-broker messages than all other alternatives while achieving at least the same communication latency.

We have published this contribution in [73–77].

1.2.3 MockFog: Automated Execution of Fog Application Experiments in the Cloud

We propose MockFog, an approach for the emulation of a fog infrastructure testbed in the cloud that can be manipulated based on a predefined orchestration schedule. In an emulated fog environment, virtual cloud machines are configured to closely mimic the real (or planned) fog infrastructure. By using basic information on network characteristics, either obtained from the production environment or based on expectations and experiences with other applications, interconnections between the emulated fog machines can be manipulated to show similar characteristics. Likewise, performance measurements from real fog machines can be used to determine resource limits on Dockerized [43] application containers. This way, fog applications and fog systems can run in the cloud while experiencing comparable performance and failure characteristics as in a real fog deployment.

With an emulated infrastructure, developers can also change machine and network characteristics, as well as the workload used during application benchmarking or testing at runtime based on an orchestration schedule. For example, this can be used to evaluate the impact of sudden machine failures or unreliable network connections as part of a system test with varying load. While testing in an emulated fog will never be as "good" as in a real production fog environment, it is certainly better than simulation-based evaluation only. Moreover, it allows application engineers to test arbitrary failure scenarios and benchmark various infrastructure options at large scale, which is also not possible on small local testbeds.

We have published this contribution in [79, 81].

1.3 Organization of this Thesis

This thesis is divided into four parts. Part I, Foundations, starts with this chapter, continues with a presentation and discussion of the scientific background, including our generally applicable geo-context model (Chapter 2), and ends with an overview of related work (Chapter 3). In Part II, Design & Implementation, we describe the approaches related to our three main contributions, corresponding system designs, and details on respective proof-of-concept implementation prototypes. First, we present BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems (Chapter 4). Second, we present GeoBroker, a single node pub/sub broker system leveraging geo-context for IoT data distribution (Chapter 5). Third, we present DisGB, an extension of the GeoBroker system that leverages geo-context to additionally improve inter-broker routing (Chapter 6). Fourth, we present MockFog, an approach for the automated execution of fog application experiments in the cloud (Chapter 7). In Part III, Experiments, we evaluate the contributions in the same order that we used in Part II. For this, we describe how we set up experiments and present experiment results (Chapters 8 to 11). In Part IV, Conclusion, we summarize the contributions (Chapter 12) and discuss our results and potential future research directions (Chapter 13). A full list of publications can be found in Appendix A. A list of the primary open-source software prototype contributions can be found in Appendix B.

Chapter 2

Background

In this chapter, we establish necessary foundations and a common level of knowledge on the scientific background. For this, we briefly define our understanding of cloud, edge, and fog in Section 2.1, before describing the vision and characteristics of the Internet of Things (IoT) in Section 2.2. In Section 2.2, we also explain why a fog computing environment is a good fit for IoT applications. Then, in Section 2.3, we summarize key concepts of the pub/sub paradigm, including the commonly used MQTT protocol and the kinds of message dissemination strategies in distributed pub/sub systems. In Section 2.4, we introduce our geo-context model from [75, 76]; the model is part of the background so that we can use its terminology to discuss related work in Chapter 3. Finally, in Section 2.5, we describe basic benchmarking & testing principles.

2.1 Cloud, Edge, and Fog

The cloud comprises a shared pool of computing resources that can be provisioned, accessed, and released on-demand via the network [121]. Building on the *everything as a service* concept [46], the cloud aims to allow developers to concentrate on developing application without having to worry about matters such as data center operation or scaling machines. For example, developers can acquire infrastructure access to virtual machines via AWS EC2 [160] or use a managed service such as AWS IoT [162]. The underlying physical infrastructure, however, might be located far away from clients, which can lead to performance issues for real-time applications [24] and also raises privacy and security concerns [192].

Edge resources such as routers, switches, or small servers, are located in physical proximity to clients and thus available with low latency. Preventing data from being transmitted via the Internet also reduces privacy and security risks. Resources are, however, not available on demand, only possess limited compute power, and not scalable without physical changes.

A fog infrastructure can comprise edge and cloud resources; it can be seen as a tree like,

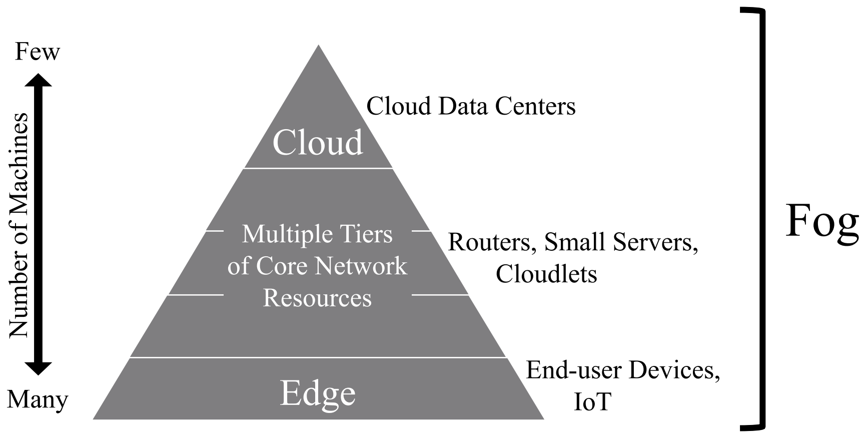


Figure 2.1: The fog comprises resources located at the edge, within the core network, and in the cloud.

distributed architecture with multiple tiers [21, 22]: On lower tiers, i.e., near or at the edge, devices often communicate without human intervention to ensure fast, low latency responses. Tasks that involve humans or require many resources, such as data analysis, visualization, or reporting, occur mostly on higher tiers on stronger machines such as cloudlets [155, 156] within the core network [17] or run directly in the cloud (see Figure 2.1). As many machines from lower tiers usually transmit data to few machines on higher tiers, one can here also access data from highly distributed sources at a single location. Still, such a distributed architecture comes with its own set of challenges. The main obstacles for broader adoption of fog computing are commonly seen as the lack of edge services and standardized hardware, increased infrastructure and quality of service (QoS) management efforts, missing network transparency, as well as the difficulty to comply with physical security, legal, and regulatory requirements [17].

2.2 The Internet of Things

Recent advances in mobile technologies and cyber-physical systems have led to a massive increase in data generation and distribution at the edge of the network. A vision of the IoT is to make this data available to heterogeneous applications and other devices [72] to enable new and better services for society and industry. For instance, efficient and intelligent communication between cars, bikes, and other road users could improve road safety [105]. By 2030, the global IoT market is expected to grow to 24.1 billion connected devices, up from 7.6 million at the end of 2019 [92]. This also comes with a massive economic impact that is estimated to grow from USD 465 billion to USD 1.5 trillion, 66% in services and 34% in hardware offerings, by 2030 [92].

While there are many proposals for an IoT architecture [106, 146], scientists have not agreed on a commonly accepted reference architecture, yet [57]. Still, existing architecture proposals have in common that IoT devices "sense and actuate physical phenomena locally" [72] and then rely on middleware systems to distribute data to other IoT devices and application services [57].

Because IoT devices only have low computational capabilities and an unreliable network connection, complex services are usually deployed in the cloud to benefit from its vast compute and storage resources. This, however, has the disadvantage that all data has to be first transmitted to the cloud.

Fog computing makes it possible to extend cloud services towards the edge [57]. As a result, IoT devices incur a lower communication delay and data can be pre-processed, aggregated, encrypted, or compressed before being transmitted to the cloud, which preserves network resources. This makes fog computing a good choice for the IoT [57, 117].

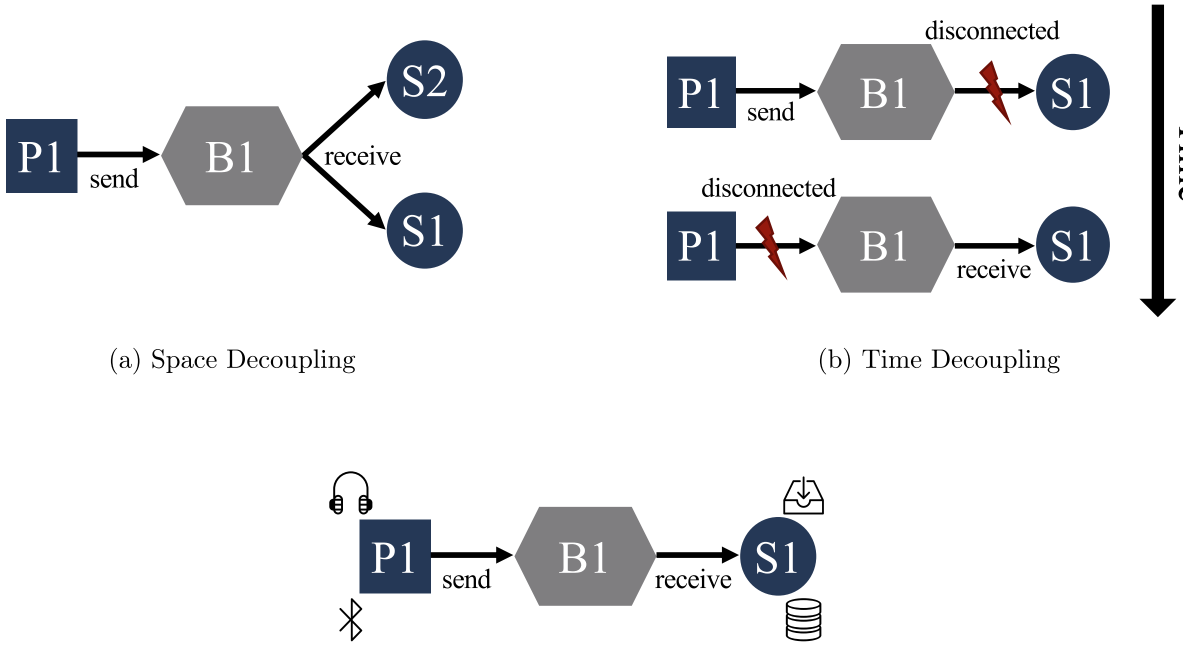
2.3 Pub/Sub-based IoT Data Distribution

An added benefit of the IoT is that data from a single device can be used by several heterogeneous applications. As pub/sub can tackle the challenge of matching device data streams and interests of data consumers [72], many IoT applications use pub/sub systems as communication middleware. Because pub/sub is a very mature research domain, there are many solutions that aim to solve different problems [154, p. 1]. In this thesis, we focus on solutions that are suited for the distribution of IoT data. In the following, we first describe general principles of the pub/sub paradigm and why it is a good fit for IoT data distribution (Section 2.3.1). Then, we introduce the MQTT protocol, a de-facto standard for pub/sub-based IoT communication platforms (Section 2.3.2.) Finally, we describe message dissemination strategies for distributed publish/subscribe systems that are suitable for fog-based deployments (Section 2.3.3).

2.3.1 The Publish/Subscribe Paradigm

The two main entities of a pub/sub system are the publishers and the subscribers [154, p.4]: Publishers are data producers, i.e., they create content by distributing events. Subscribers are data consumers, i.e., they express their interest in certain content through subscriptions.

In a very basic setup, publishers and subscribers connect to a single, centralized broker that handles communication and matching events and subscriptions: If the content of an event matches a particular subscription content filter, the check is successful and the event is delivered [154, p. 148f.]. In the following, we refer to this kind of check as *ContentCheck*. The expressiveness of a ContentCheck depends on the matching type: Channel-based ContentChecks provide the lowest level of expressiveness while content-based ContentChecks provide the highest level of expressiveness [154, p. 149]: For the former, an event is published to a specific channel, all subscriptions that target this channel will pass the ContentCheck. For the latter, parts of the event itself are used for the check, e.g., an event with the content $temperature = 30°C$ will pass the ContentCheck for a subscription with the filter $temperature > 25°C$. Having such a high level of expressiveness can be a significant downside for IoT applications because publishers



(a) Space Decoupling

(b) Time Decoupling

(c) Synchronization Decoupling

Figure 2.2: Three examples with a single broker (B1), one publisher (P1) and one or two subscribers (S1, S2) to showcase the types of decoupling in broker-based pub/sub systems [50].

and subscribers must agree on content formatting. Therefore, large commercial systems such as AWS IoT [162] rely on MQTT [7] which uses a topic-based ContentCheck; the content of the message itself, however, can be arbitrarily formatted (see also Section 2.3.2).

The goal of the pub/sub paradigm is to connect clients across space, time, and synchronization [50]; properties that make pub/sub very appealing for IoT applications. We explain these three types of decoupling with the help of the examples in Figure 2.2:

Space decoupling Publishers and subscribers are not aware of each other, nor do they need to know the location of other clients or how many clients receive a given event. In the example, P1 does not know that its events are received by the two subscribers S1 and S2. Furthermore, S1 and S2 do not know whether another subscriber receives the events they receive.

Time decoupling Publishers and subscribers do not need to be online at the same time. In the example, P1 publishes an event and S1 receives it as soon as it connects to the broker, even if P1 has disconnected in the meantime.

Synchronization decoupling Publishers and subscribers receive and send events asynchronously. In the example, P1 and S1 can do other things while waiting to send/receive events.

Depending on the specific application use case, however, not all properties are always needed. Especially the time decoupling property is often relaxed to reduce the number of buffered events.

In general, most pub/sub systems can be classified into two categories: broker-based or P2P-based (peer-to-peer). Broker-based solutions rely on a set of interconnected brokers[1] for distributing messages [118]. Clients, i.e., the publishers and subscribers, do not interact directly; instead, they connect to one of the brokers that handles communication. This approach's main benefit is that brokers handle the compute-intensive task of distributing and matching events and subscriptions. Furthermore, clients can temporarily disconnect from the broker network and receive messages after re-connecting. These two characteristics make broker-based solutions a good fit for IoT applications: IoT devices usually have low computational power and unreliable network connections.

In P2P-based solutions, there are no brokers; clients have to route messages themselves [118]. Therefore, each client has to do at least some part of the event matching process. When delivering events to other clients, clients also partly violate the space decoupling property. Furthermore, for the time decoupling property, clients may have to temporarily store events for other clients that are currently disconnected. While P2P-based solutions can save cost for smaller setups as no dedicated communication components have to be set up and maintained, the approach is often not feasible for the IoT due to limited resources at the edge. Thus, for the remainder of this thesis, we focus on broker-based solutions.

2.3.2 MQTT Pub/Sub Protocol

MQTT is a lightweight, open, and simple pub/sub protocol that works well in constrained environments such as the IoT where a small code footprint is required and compute/network resources are limited [7]. It has evolved to a de-facto standard for pub/sub-based IoT communication platforms [57, 72, 142].

MQTT-based systems are broker-based and topic-based, so clients connect to brokers and create subscriptions based on a topic tree. Clients then publish events to topics which the broker uses to identify matching subscriptions. Topics are identified by their names, which may consist of multiple levels separated by "/". For example, if a subscribers creates a subscription for the topic a/b, it will receive all events published to the topic a/b, but no other events, e.g., published to topic a/c (see also Table 2.1).

Besides such fixed topics, clients can also use special wildcard characters to subscribe to multiple topics at once. A wildcard is either valid at a single topic level ("+") or at multiple topic levels ("#"). For example, the subscription topic $+/b/+$ matches the event topics $a/b/b$ and $a/b/c$, but not a/c. However, the event topics $a/b/b$, $a/b/c$, a/c, and all other event topics starting with a match the subscription topic $a/\#$ (see also Table 2.1).

[1] Or on a single broker as in the example shown in Figure 2.2.

Table 2.1: Examples of matching (✓) and not matching (✗) subscription and event topics.

	Event Topic				
Subscription Topic	a/b	a/c	a/b/b	a/b/c	b/b/b
a/b	✓	✗	✗	✗	✗
a/b/c	✗	✗	✗	✓	✗
+/b/+	✗	✗	✓	✓	✓
a/#	✓	✓	✓	✓	✗

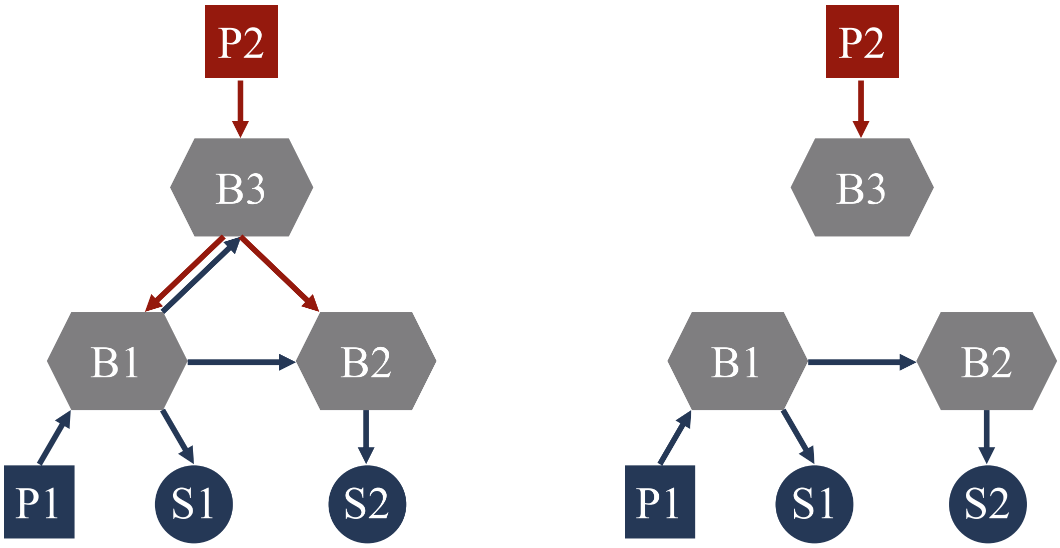
2.3.3 Message Dissemination in Distributed Publish/Subscribe Systems

In distributed deployments, e.g., when using a fog infrastructure, brokers are deployed at multiple sites. Client devices connect to the nearest broker, to which we refer to as *local broker* (LB), for low latency communication with other devices in physical proximity. The brokers route messages to other brokers so that devices that are not connected to the same broker can communicate. There are various inter-broker routing strategy options which can be classified into the three categories *flooding*, *gossiping*, and *selective* [6, 154].

Flooding

When flooding events, brokers broadcast the events they receive from local publishers to all other brokers (Figure 2.3a). As a consequence, the number of inter-broker message hops is one when a matching subscriber is connected to another broker (P1 and S2), and zero, when a subscriber is connected to the same broker as the publisher (P1 and S1). A downside of this strategy is that brokers often receive events for which no matching subscriber exists. In the example, there is no subscriber that matches the red events of P2. Thus, this strategy minimizes end-to-end latency but leads to excess data, i.e., data that is not needed by any subscriber and subsequently dropped by brokers.

When flooding subscriptions, brokers broadcast the subscriptions they receive from local subscribers to all other brokers. This way, every broker knows about all subscriptions and only distributes events to another brokers if there is a matching subscriber (Figure 2.3b). In the example, B3 does not distribute the red events of P2, and B1 only forwards the blue events of P1 to B2. Similarly to flooding events, the number of inter-broker messages hops is also one or zero. Thus, this strategy also minimizes end-to-end latency while not distributing events to brokers without matching subscribers. One has to keep in mind, however, that subscriptions (and their updates/removals) are broadcasted to all brokers. This leads to a lot of excess data if there are many brokers and subscription updates, but only few events.



(a) When flooding events, brokers send events of local publishers to all other brokers.

(b) When flooding subscriptions, brokers send events of local publishers only to brokers with matching subscriptions.

Figure 2.3: Illustrative example with three brokers (B1, B2, B3), a publisher with events that match the subscriptions of two subscribers (P1, S1, S2), and a publisher with events that match no subscriptions (P2).

In conclusion, flooding ensures minimum end-to-end latency. It does, however, not scale well as every broker has to process all events or subscriptions from every other broker which also leads to a high network load. Flooding solutions are therefore more commonly used as a reference for comparison, e.g., in [53, 118], rather than being used in real setups.

Gossiping

With gossiping, brokers distribute messages to a subset of brokers based on a probability distribution [154, p. 73]. The main advantage of systems that build on gossiping is their tolerance for very dynamic environments. However, depending on the probability distribution, messages might incur very high delivery latency or might not arrive at some brokers. While IoT devices might operate in a very dynamic environment, the brokers, which use the routing approach, do not. Instead, it is more likely that the brokers are deployed in a limited number of (high tier) fog regions with low churn rates. This makes gossiping an uncommon routing strategy for IoT applications or fog deployments. Commonly known gossiping pub/sub systems that are used for other purposes include SpiderCast [35], PolderCast [165], and Vitis [137].

Selective

Selective strategies are either filter-based or build upon rendezvous points (RP) [6]. For filter-based strategies, similarly to the flooding subscriptions strategy, subscription information has to

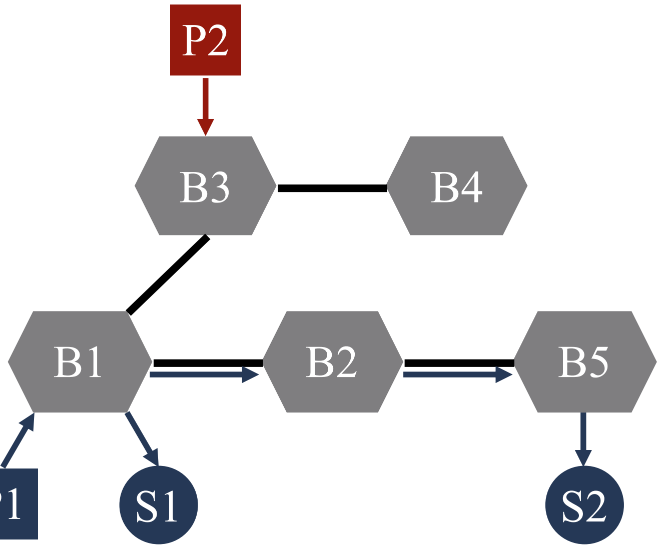


Figure 2.4: For filter-based strategies, events are only forwarded to brokers that lie on a path leading to machting subscribers.

be available at all brokers. In network configurations where not all brokers are directly interconnected, however, brokers only exchange subscription information with their direct neighbors[2] which reduces communication overhead compared to flooding. Subscription dissemination can be further optimized as done in [29, 42, 190]. Events are only forwarded to brokers that lie on a path leading to matching subscribers [6]. To determine this path, brokers match incoming events with all subscriptions of neighboring brokers and forward the event accordingly. Events have to traverse multiple brokers rather than being sent directly, however, which can result in high end-to-end latency. In the example shown in Figure 2.4, events from P1 match the subscriptions of S1 and S2. Thus, B1 delivers the event to S2 and forwards it to B2, i.e., towards S2. B2 matches the event again and forwards it to B5, which then again runs through the matching process and delivers it to S2. Events from P2, on the other hand, are not distributed to any broker as B3 is not aware of any matching subscribers.

RPs reduce communication cost by being a "meeting point" for subscriptions and events: the matching occurs at the RP brokers [154, p. 166]. Hence, they limit the propagation of events or subscriptions to a small subset of brokers; this improves system efficiency. The main challenge with RP-based strategies is selecting the RP for a given event/subscription. If the RP is close to the publishers or subscribers of an event, the end-to-end latency is comparable to the one of flooding solutions. If, however, the RP is located far away from the publisher and the subscribers, the end-to-end latency can become very high. In the example shown in Figure 2.5, the RP for blue events and subscriptions is B1. This means that the subscriptions of S1 and S2, as well as all events from P1, are sent to B1 for matching. B1 then delivers events to S1 directly and forwards them to B5 for delivery to S2. As blue events and subscriptions are only routed to the LB of clients, i.e., the local brokers to which publishers and subscribers connect,

[2] Neighborhood relationships do not necessarily rely on physical connections but can be based on knowledge, i.e., defined by an administrator or based on technical constraints [6].

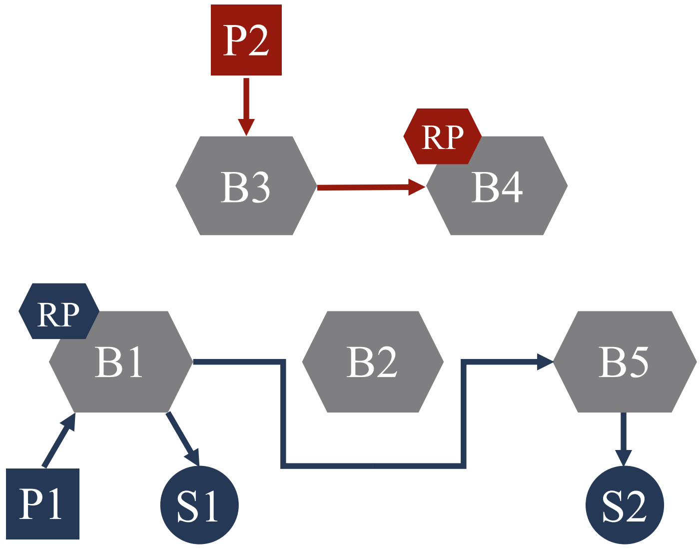


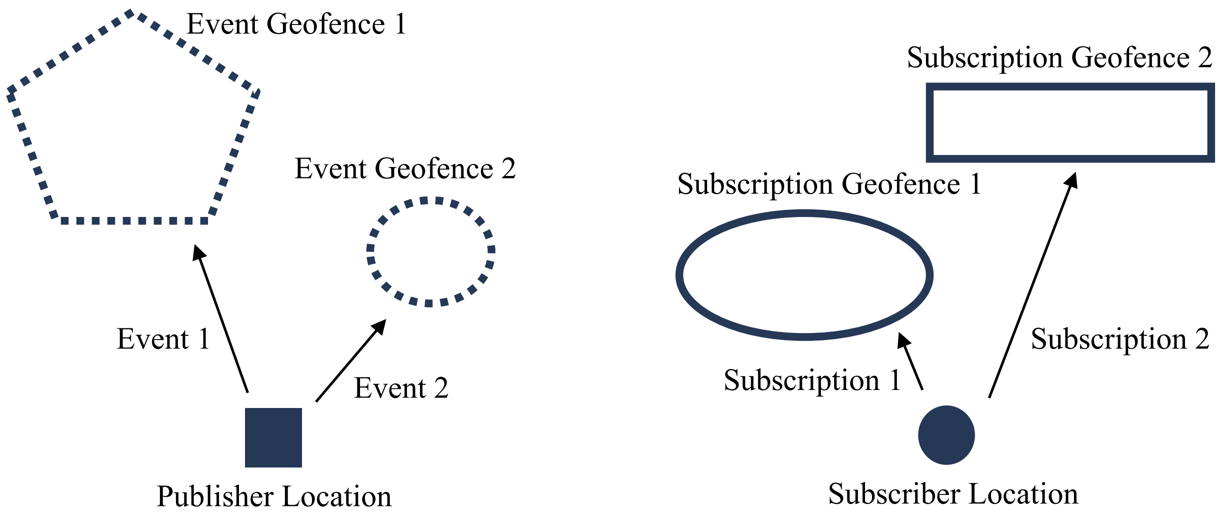
Figure 2.5: The end-to-end latency for RP-based strategies depends on the selected RP. B1 is a good RP for blue messages, as events are only routed via LB of involved clients.

the event delivery latency is minimal and comparable to flooding. The RP for red events and subscriptions is B4. This means that all events of P2 have to be routed to B4, even though no matching subscriptions exist. If there were a matching subscription, e.g., from a client connected to B2, events would still be routed via B4 even though B4 is not the LB of any involved client. Therefore, communication latency is higher than the one of a flooding solution. There are many RP-based pub/sub systems, e.g., Scribe [147], Hermes [134], and Meghodoot [68], as well as [9, 124, 178]. The strategies we propose in this thesis for distributing IoT data also build upon RP-based routing; we specifically focus on selecting RPs that are close to either the publisher or the subscribers of an event to minimize end-to-end latency.

2.4 Four Dimensions of Geo-Context

Previous work has already proposed to use geo-context information for more advanced control of data distribution. The current focus, however, has not been on developing a general view on the geo-context of IoT devices. Instead, the authors typically designed a system for a particular use case in which location-based data needs to be processed; thus, they do not consider all geo-context dimensions but rather only those relevant to their specific use case. Based on the available related work, we derived a generally applicable definition of the entire geo-context. In the following, we first present this definition so that we can use the terminology to discuss which dimensions have been considered in related work (Section 3.2).

We identified four geo-context dimensions (see Figure 2.6). Clients, i.e., publishers and subscribers, have a geographic location (**publisher location** and **subscriber location**), which consists of a latitude and a longitude value. For stationary clients, such locations may already



(a) A publisher has a location and each event targets a certain area.

(b) A subscriber has a location and each subscription targets a certain area.

Figure 2.6: We identified four geo-context dimensions.

be known and provided as a parameter when starting the client. In all other cases, one can, for example, determine the location using GPS, WiFi trilateration [148], Ultra-Wideband distance sensors [125], or even based on smartphones inertial sensors [188]. There are also various optimization algorithms that improve accuracy and reduce resource consumption [101]. This is particularly important for constrained IoT devices.

Beyond this, publishers and subscribers each have an area of interest; we propose to use geofences to describe these areas. A Geofence is a virtual fence surrounding a geographical area[3]. For our purposes, a geofence can have an arbitrary shape and may comprise non-adjacent subareas, e.g., Germany and Italy. One can specify geofences in the `Well-Known Text` [114] format. The **event geofence** can be unique for each individual event. It ensures that only subscribers located in the specified area receive the event, i.e., subscriber locations must be inside the event geofence. The **subscription geofence** can be unique for each individual subscription. It ensures that only the events of publishers located in the specified area may be delivered to the subscriber, i.e., publisher locations must be inside the subscription geofence.

In Chapter 5, we describe in detail how to use these four geo-context dimensions during the event matching process of a pub/sub system.

[3] As a general usage example, Reclus and Drouard describe a scenario in which such fences are used to notify factory workers about approaching trucks [143].

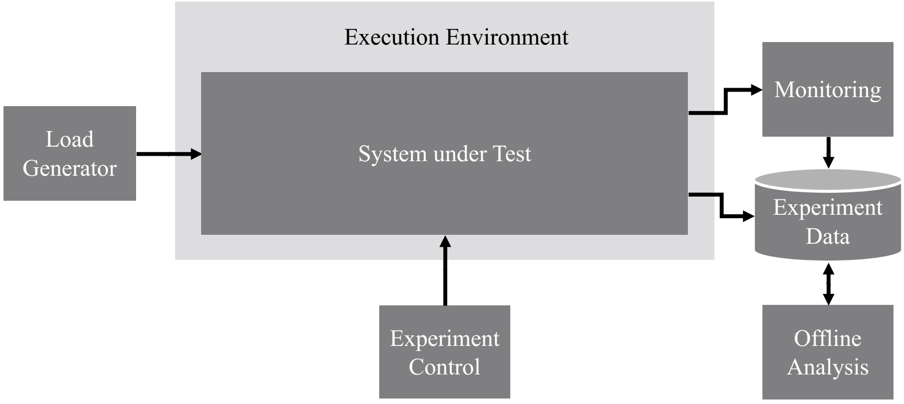


Figure 2.7: Standard components of cloud service benchmarking as pictured by Bermbach et al. [18, p. 15], includes minor adaptions.

2.5 Benchmarking & Testing

Benchmarking, testing, and monitoring are three important concepts for evaluating software and infrastructure options according to specific quality metrics such as performance, fault-tolerance, or data consistency [18]. The difference between the concepts is that monitoring achieves this through passively observing a production system while testing and benchmarking actively stress a system under test that has been specifically set up for this purpose.

Brogi et al. [26] consider multiple fog-specific challenges such as "limited hardware resources and unstable connectivity at the edge, platform heterogeneity, and node failures" for their fog monitoring system FogMon. While these challenges also exist for testing and benchmarking, however, one has to additionally deal with setting up testing infrastructure and rolling out the system under test. Our third contribution MockFog (see Section 1.2.3) specifically aims to simplify this task for developers. In the following, we introduce basic benchmarking (Section 2.5.1) and testing (Section 2.5.2) concepts and highlight fog-specific challenges.

2.5.1 Benchmarking

Benchmarks are typically run to answer a specific question for a system (or application) under test [18, p. 14]. Since they need to generate meaningful results quickly, they usually substantially affect the evaluated system. Thus, benchmarking is typically not done in production environments. Furthermore, benchmarking requires a high degree of control, e.g., for reproducibility reasons.

Figure 2.7 shows the standard components of cloud service benchmarking as pictured by Bermbach et al. [18, p. 15]. The system under test is set up within an execution environment that resembles its production environment. During the benchmarking process, a load generator creates a synthetic system load and experiment data is collected. One can also use monitoring systems similar to those used within the production environment to complement the experiment data collection. There is usually also an experiment control component that can manipulate the system under test at runtime for more advanced benchmarking scenarios. Finally, when the benchmarking process terminates, the collected experiment data is analyzed.

While the basic components for fog application benchmarking are similar, setting up the execution environment and managing the system under test and load generation is more complex than the ease of adoption developers are used to from the cloud [17]: A fog execution environment comprises physical machines that are located at different sites. In difference to a cloud-based environment, one can therefore not merely spawn virtual machines but has to set up a physical infrastructure with similar network and compute characteristics. Furthermore, rolling out and managing the system under test is more complicated and error-prone since not all sites necessarily have a good network connection. This can also be a problem when beginning workload generation: even if the system under test does not run on the production machines, the production system might still be affected if the systems share network resources. Our third contribution MockFog (see Section 1.2.3) specifically addresses such problems.

Another issue with fog application benchmarking is the lack of standardized benchmarks. While there are numerous benchmarks available for cloud services, e.g., YCSB [38], DCBench [95], or CloudRank-D [115], there is a surprising lack of similar benchmarks for fog applications and systems. We are only aware of a single contribution in this area: DeFog [120]. In this work, the authors also explain that part of the problem is that "there is a limited understanding of the real workloads that can benefit the most from using fog computing", which is why no fog benchmarks are available yet. As a first step towards creating a portfolio of re-useable fog benchmarks, DeFog comprises six benchmarks with different characteristics. Still, when testing more complex fog applications, we believe that it is still best to create customized **application-driven benchmarks** [18, p. 67] when studying quality aspects specific to the application or system under test.

2.5.2 Testing

Testing is done to ensure a high software quality [23, p. 3]. For this, we need to (1) judge the current quality of a given software system and (2) discover existing problems [96, p. 3].

Testing can be done on different abstraction levels. The most commonly used abstraction levels are system testing, integration testing, and unit testing [96, p. 12]. On the highest level of abstraction, **system testing** determines whether the software meets its specified requirements [1, p. 6]. For this, it builds upon the assumption that software components work individually to test how the complete system behaves for a given input. One level below, **integration testing** determines whether software components can communicate correctly via their interfaces [186]. A well-known example that highlights the importance of integration testing is the Mars Climate Orbiter mission that failed in September 1999 [96, p. 229]: two software components exchanged acceleration data in different units of measurement (imperial units and metric units). Finally,

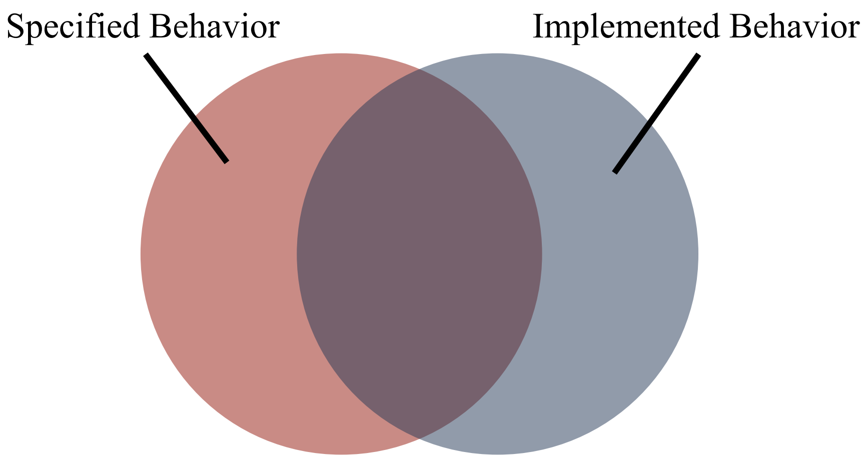


Figure 2.8: Interplay between specified software behavior and implemented software behavior based on [96, p. 6].

on the lowest level, **unit testing** is used to determine whether individual software "units", such as Java methods, behave as expected [1, p. 7].

To better understand how to create individual test cases for any of these three abstraction levels, we must first understand the two types of software behavior: specified behavior (what do we expect) and implemented behavior (what do we get). Jorgenson [96, p. 6] uses a Venn diagram to highlight the interplay between these two types, see also Figure 2.8. To increase the software quality, we need to determine and maximize the overlapping degree (violet area). In other words, we need to minimize behavior that is specified but not implemented (red area) and behavior that is implemented but not specified (blue area), e.g., software bugs.

For this purpose, developers can use specification-based testing and code-based testing to create test cases [96, p. 6ff.]. To the former, some authors also refer to as functional testing or black-box testing [183] since one only uses the software/requirement specification for the identification of test cases. Specification-based testing has two advantages: First, tests are decoupled from the actual implementation and remain valid even if the implementation changes. Second, tests can be developed from the very beginning and in parallel to the implementation. However, since tests only build upon the specification, they cannot identify behavior that has been implemented but is not specified (blue area in Figure 2.8). In contrast, for code-based testing, developers use the actual program code for test creation. Thus, some authors refer to it as white-box testing or clear-box testing [183]. Building only upon the software code, however, such tests cannot identify behavior that is specified but not implemented (red area in Figure 2.8). Jorgenson therefore concludes that combining both approaches is the best way to increase software quality [96, p. 9].

In software engineering, tasks can be grouped into two categories: revenue tasks and excise tasks [1, p. 10]. Revenue tasks contribute something to the solution of a problem, e.g., determining a test case, while excise tasks are necessary but do not contribute something to the software behavior, e.g., running a test case. Thus, doing excise tasks is uneventful and should be supported by automation as much as possible to reduce mistakes and free up time. Since unit tests tend to evaluate small isolated features only, similar tools can be used for fog-based and cloud-based applications. Integration and system tests, however, usually require access to a testing infrastructure that resembles the production environment. Creating, maintaining, and tearing down such an infrastructure for fog-based applications is a labor-intensive excise task; we aim to automate this process as much as possible with our third contribution MockFog (see Section 1.2.3).

Chapter 3

Related Work

In this chapter, we discuss related work to our three contributions (see Sections 1.2.1 to 1.2.3). In Section 3.1, we review related work on routing in distributed pub/sub systems. The work presented here does not consider geo-context dimensions, so it is most closely related to BC-Groups, but also to GeoBroker & DisGB. Then, in Section 3.2, we discuss related work on data distribution leveraging geo-context. While this includes pub/sub systems similar to GeoBroker & DisGB, we also discuss a broader selection of geo-context-aware data distribution approaches. Finally, in Section 3.3, we discuss related work to MockFog, i.e., on fog application testing & benchmarking.

3.1 Distributed Pub/Sub Systems

Efficient routing of events and subscriptions in geo-distributed pub/sub systems is a mature research domain, and there are many surveys that summarize the state of the art, e.g., [6, 15]. Still, only few approaches use IoT-specific domain knowledge or are explicitly built for distributed environments with similar characteristics as the fog.

In Section 3.1.1, we present an overview of inter-broker routing approaches that build upon RPs. Then, in Section 3.1.2, we present an overview of approaches that build upon other inter-broker routing strategies.

3.1.1 Approaches with Rendezvous Points

There are some related approaches that also use IoT-specific domain knowledge for the selection of RPs:

An et al. [2] propose PubSubCoord, which, at first glance, looks very similar to BCGroups. They also group local brokers for low latency communication between physically close clients.

There are, however, two key differences. First, since group membership depends on network segments, they cannot control group sizes. BCGroups controls the group sizes to manage the latency and excess data dissemination tradeoff. Second, coordination and message exchange between local groups relies on Zookeeper and a custom broker implementation that serves as RP. BCGroups, on the other hand, is specifically designed to be used in conjunction with existing cloud-based pub/sub offers.

Cao and Singh [28] propose MEDYM and H-MEDYM. In MEDYM, brokers need to know about all other brokers in the network and the sum of their subscriptions, which is infeasible in large deployments. To circumvent this limitation, H-MEDYM also proposes to create broker groups. In each group, a so-called matcher broker handles the communication with the matcher brokers of other groups. Within a group events must also be routed via the group's matcher. This can increase efficiency but also increases latency and leads to a single point of failure, even for local communication. Furthermore, matcher brokers must be aware of other matcher brokers and their subscriptions, which is not feasible for large fog deployments.

Banno et al. [9] propose to use a skip-graph to select RPs that satisfy the "strong relay-free" property to prevent data distribution of events that do not have any subscriber. Teranishi et al. [178] extend this approach to improve locality awareness. With this, both author groups specifically take into account IoT characteristics. However, the skip-graph is a shared data structure that must comprise all publishers and subscribers. Thus, keeping this structure consistent across machines in large fog deployments is infeasible.

Furthermore, there are also related approaches that are built specifically for distributed environments with similar characteristics as the fog:

Rausch et al. [142] propose a fog-enabled geo-distributed broker. To this end, they use a centralized cloud service that continuously orchestrates brokers and migrates MQTT clients. The cloud service, however, needs a comprehensive global view on inter-node latency, edge brokers, clients, and subscriptions to select RPs. Keeping this view up to date can be challenging in volatile deployments. Moreover, migration might lead to message loss, which is especially problematic in scenarios with mobile end-devices.

Nguyen et al. [124] propose a communication middleware that combines a big data management system (BDMS) with a distributed pub/sub broker network. By routing all events through the cloud BDMS, their system can process a large number of events before efficiently delivering them to matching subscribers through the fog-based broker network. Still, physically close clients cannot communicate directly but have to communicate via the cloud BDMS. Thus, communication latency might be too high for some IoT scenarios, and there can be availability problems in the presence of network partitions.

Finally, most RPs-based routing strategies do neither consider the characteristics of IoT applications or the fog environment:

For example, Kawaguchi and Bandai [102] propose a distributed broker system in which the RP is identified by a custom topic structure which embeds geographic information. This approach does not work well when some clients are located far away from the broker responsible for one of their topics due to high round-trip latency. Gascon-Samson et al. [59] propose MultiPub, which optimizes routing between cloud regions to minimize cloud-based costs considering fixed QoS guarantees. With the IoT, this approach can run into scalability issues as routing is optimized by a centralized controller that needs to be aware of all clients, events, and subscriptions.

Approaches that are built upon distributed hash tables such as Scribe [147], Hermes [134], Meghdoot [68], or Dynatops [193] are also not a good fit for a fog environment since they assume symmetric networks while the fog is organized in a tree-like structure. While these approaches can be very efficient in terms of required inter-broker messages, they might route messages across any node even though individual nodes might not have access to sufficient compute or networking resources. For the fog, however, this is not acceptable due to the heterogeneity of machines as well as partly unstable and relatively slow network connections. This is also the issue with grid-based approaches such as the one presented in [177].

3.1.2 Other Routing Approaches

This section discusses related work that does not build upon RP for the routing of events and subscriptions.

Shun et al. [158] propose a topic-based fog computing architecture which they use for the exchange of semantically enhanced Internet of Vehicles (IoV) data. In contrast to our approaches, they use subscription flooding (see Section 2.3.3), resulting in limited scalability and potentially overloaded brokers with few resources.

Banno et al. [8] propose to interconnect heterogeneous and distributed MQTT brokers through an additional middleware layer between brokers and end-devices. This layer also takes care of distributing messages between brokers based on customizable routing strategies. For their paper, they only implemented flooding. We assume that adding, for example, our BCGroups strategy is possible as well. Unfortunately, their source code is not publicly available, so we could not verify this.

Zeidler et al. [53, 190] aim to add mobility awareness to Rebeca [128]. For this, they buffer all events relevant to a client at the last broker at which the client has been connected. Then, when a client reconnects at another broker, it can receive missed events. Their approach builds on selective filtering (see Section 2.3.3) rather than RPs.

Similar to the RP-based related work that we presented in the previous section, there are also a multitude of other system designs that are not specifically built for the fog or the IoT, e.g., [29, 40, 41, 165, 167]. These systems are highly optimized for a variety of execution environments and purposes; including P2P environments [165] and distributing vast data volumes at

Facebook [167]. Still, they do not consider the specific characteristics of fog environments or the IoT when distributing data. This also applies to approaches such as [12, 58, 103], which are tailored only for the cloud and thus assume LAN connectivity between machines. Therefore, they cannot cope with geo-distribution.

3.2 Geo-Context Information

One can use spatial information for various purposes, e.g., to optimize replica placement [189]. For this related work discussion, we concentrate on approaches related to data distribution, i.e., related to GeoBroker & DisGB (see Section 1.2.2). Table 3.1 summarizes which of the four geo-context dimensions are considered by the discussed related work; we use our geo-context dimension terminology from 2.4 to improve comparability.

Chen et al. [34] propose a centralized pub/sub system that delivers events to clients when they enter "zones" defined by publishers, i.e., event geofences. While this allows publishers to control event distribution based on areas they consider as relevant, subscribers cannot control data distribution with subscription geofences.

The authors of [27, 36, 66, 67, 109] consider the subscription geofence and the publisher location, i.e., the geo-context dimensions needed for the subscription GeoCheck. However, neither group of authors lets publishers control the matching of events based on event geofences and subscriber locations (event GeoCheck): Bryce et al. [27] propose MQTT-G, an extension of the MQTT protocol with Geolocation. While subscribers can define subscription geofences to control event distribution, their geofences are only created once per subscriber rather than for individual subscriptions. Chow et al. [36] present GeoSocialDB, a system that provides three location-based services for social networks: a news feed, news ranking, and recommendations. Each of these services could be queried to retrieve all data of publishers, such as previously published events of publishers located (publisher location) in an area defined by a subscriber (subscription geofence). Guo et al. [66, 67] propose a location-aware pub/sub service that delivers events based on subscription geofences attached to subscriptions and publisher locations.. Li et al. [109] propose to use an R-Tree index structure to efficiently identify which publishers are located in areas defined by subscribers. Again, these groups of authors only look at geo-context from one perspective, so their approaches do not work with event geofences and subscriber locations.

Wang et al. [184] propose the AP-Tree to efficiently support location-aware pub/sub. While they only discuss the publisher location and subscription geofence in their paper, they state that using an event geofence could also be possible with their approach. They do not, however, consider subscriber locations.

There are also a few publications that propose to use geo-context information to improve inter-broker routing, similarly to DisGB:

Table 3.1: An overview of the geo-context dimensions considered by related work.

	Location		Geofence	
Related Work	Subscriber	Publisher	Subscription	Events
Chen et al. [34]	Yes	No	No	Yes
Bryce et al. [27]	No	Yes	Yes	No
Chow et al. [36]	No	Yes	Yes	No
Guo et al. [66, 67]	No	Yes	Yes	No
Li et al. [109]	No	Yes	Yes	No
Wang et al. [184]	No	Yes	Yes	Partially
Chapuis et al. [32, 33]	Partially	Partially	Yes	Yes
Cugola & Munoz de Cote [42]	Yes	Yes	Yes	Yes
Frey & Roman [56]	Yes	Yes	Yes	Yes
Herle et al. [87, 88]	Yes	Yes	Yes	Yes

Chapuis et al. [32, 33] propose a horizontally scalable pub/sub architecture that supports matching based on a circular geofence around publishers and around subscribers. Hence, their event matching does not consider client locations independent of geofences. Furthermore, their geofences can only be circular while we aim to support arbitrarily shaped geofences. They also specifically aim to enable horizontal scalability of clustered machines and do not support geo-distributed deployments.

Cugola and Munoz de Cote [42] use all four geo-context dimensions for the routing of events and subscriptions. Their approach, however, builds upon selective filtering rather than rendezvous points; so events must traverse a multi-cast tree, which increases end-to-end latency. Furthermore, it is also impossible to directly connect all brokers because each broker needs to know the current location of all clients connected to neighboring brokers. This prevents scaling to large amounts of mobile clients in fully meshed environments.

Frey and Roman [56] propose an P2P-based event routing strategy that distributes events to hosts within a defined *context of relevance* (comparable to our event geofence), if the publishing host is located within a defined *context of interest* (comparable to our subscription geofence). For their approach to work, however, clients must store and share information on events, existing subscriptions and neighboring hosts, so it is not suited for applications with many IoT devices.

Herle et al. [87, 88] propose to extend the MQTT protocol so that events can be matched based on locations and geofences appended to events and subscriptions. Each subscription or event, however, can have either a geofence or a location. Thus, it is only possible to do one of the two DisGB GeoChecks, but never both. They also do not propose to additionally improve inter-broker message routing.

3.3 Fog Application Testing & Benchmarking

Testing and benchmarking distributed applications in fog computing environments can be very expensive as the provisioning and management of needed hardware is costly. Thus, in recent years, many approaches have been proposed that aim to enable experiments of distributed applications or services without the need for access to fog devices, especially ones located at the edge. In the following, we briefly introduce these approaches and highlight differences to MockFog.

There are several approaches that, similarly to MockFog, aim to provide an easy-to-use solution for experiment orchestration on emulated testbeds. WoTbench [85, 86] can emulate a large number of Web of Things devices on a single multicore server. As such, it is designed for experiments involving many power-constrained devices and cannot run experiments with many resource-intensive application components such as distributed application backends. D-Cloud [10, 71] is a software testing framework that uses virtual machines in the cloud for failure testing of distributed systems. However, D-Cloud is not suited for evaluating fog applications as users cannot control network properties such as the latency between two machines. Héctor [13] is a framework for automated testing of IoT applications on a testbed that comprises physical machines and a single virtual machine host. Having only a single host for virtual machines significantly limits scalability. Furthermore, the authors only mention the possibility of experiment orchestration based on an "experiment definition" but do not provide more details. Marvis [14] builds upon Héctor and aims to also integrate simulators and other emulators to create a "staging environment for the Internet of Things". Marvis seems to be at a very early stage, but in the future it might be an option to also integrate MockFog. Balasubramanian et al. [5] and Eisele et al. [49] also present testing approaches that build upon physical hardware for each node rather than more flexible virtual machines. EMU-IoT [139] is a platform for the creation of large scale IoT networks. The platform can also orchestrate customizable experiments and has been used to monitor IoT traffic for the prediction of machine resource utilization [140]. EMU-IoT focuses on modeling and analyzing IoT networks; it cannot manipulate application components or the underlying runtime infrastructure.

Gupta et al. presented iFogSim [69], a toolkit to evaluate placement strategies for independent application services on machines distributed across the fog. In contrast to our solution, iFogSim uses simulation to predict system behavior and, thus, to identify good placement decisions. While this is useful in early development stages, simulation-based approaches cannot be used to test real application components, which we support with MockFog. In [25, 108, 149], multiple other system designs for the simulation of complex IoT scenarios with thousands of IoT devices are proposed. Additionally, network delays and failure rates can be defined to model a realistic, geo-distributed system. More simulation approaches include FogExplorer [83, 84], which aims to find good fog application designs, or Cisco's PacketTracer [37], which simulates complex networks. However, all these simulation approaches do not support experiments with unmodified application components. This also applies to SimGrid [30], a widely used framework for the simulation of distributed computer systems. What makes SimGrid unique is its capability of simulating the communication between real application components that adhere to the MPI protocol by re-compiling them with a special toolkit.

Multiple groups of authors [3, 39, 55, 119, 129, 130] build on the network emulators MiniNet [127] and MaxiNet [185]. While they target a similar use case as MockFog, their focus is not on application testing and benchmarking but on network design (e.g., network function virtualization). Based on the papers, the prototypes also appear to be designed for single machine deployment – which limits scalability – while MockFog is specifically designed for distributed deployment. Finally, neither of these approaches appears to support experiment orchestration or the injection of failures. Missing support for experiment orchestration is also a key difference between MockFog and MAMMOTH [113], a large scale IoT emulator, Distem [153], a tool for building experiment testbeds with Linux containers, and EmuEdge [191], and edge computing emulator that supports network replay.

OMF [138], MAGI [91], and NEPI [136] can orchestrate experiments using existing physical testbeds. On a high level, these solutions aim to provide similar experiment orchestration capabilities as MockFog.

For failure testing, Netflix has released Chaos Monkey [180] as open source[4]. Chaos Monkey randomly terminates virtual machines and containers running in the cloud. This approach's intuition is that failures will occur much more frequently, so engineers are encouraged to aim for resilience. Chaos Monkey does not provide the runtime infrastructure as we do, but it would complement our approach very well. For instance, Chaos Monkey could be integrated into MockFog to extend its experiment orchestration capabilities. Another solution that complements MockFog is DeFog [120]. DeFog comprises six Dockerized benchmarks that can run on edge or cloud resources. From the MockFog point of view, these benchmark containers are workload generating application components, i.e., load generators. Thus, they could be managed and deployed by MockFog. Gandalf [110] is solely a monitoring solution for cloud deployments. Azure, Microsoft's cloud service offer, uses Gandalf in production. It can therefore also complement MockFog's data collection during experiment orchestration. Finally, MockFog can be used to evaluate and experiment with fog computing frameworks such as FogFrame [171] or URMILA [168].

[4] https://github.com/Netflix/chaosmonkey

Part II

Design & Implementation

In this part, we describe the approaches related to our three main contributions, corresponding system designs, and details on respective proof-of-concept implementation prototypes. First, we present BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems (Chapter 4). Second, we present GeoBroker, a single node pub/sub broker system leveraging geo-context for IoT data distribution (Chapter 5). Third, we present DisGB, an extension of the GeoBroker system that leverages geo-context to additionally improve inter-broker routing (Chapter 6). Fourth, we present MockFog, an approach for the automated execution of fog application experiments in the cloud (Chapter 7).

Chapter 4

BCGroups: An Inter-Broker Routing Strategy for the Fog

In this chapter, we present BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems. The main idea behind BCGroups is to split the set of fog brokers into well connected **broadcast groups** which use flooding (see Section 2.3.3) for intra-group communication and a cloud RP (see Section 2.3.3) for inter-group communication. Event flooding minimizes communication latency of client devices in physical proximity, i.e., where a low communication latency is often required in IoT scenarios [56, 123, 150, 158]. As a side effect, flooding also handles frequently updated subscriptions of mobile clients particularly well, as events are sent preemptively to all brokers at which a client could create its subscriptions. This, however, can also result in excess data if no matching subscriber is connected to some of the receiving brokers.

For global communication, each broadcast group elects a leader that communicates on behalf of the group with the cloud RP and through it with other group leaders. Using a cloud RP for global communication minimizes excess data. As we cannot ensure that the cloud RP is close to the publisher or subscribers of an event, however, this can increase end-to-end latency. Still, the latency requirements of clients that do not operate in physical proximity are usually less demanding. With the broadcast group size, we can control the number of flooded events and hence manage the tradeoff between excess data and latency.

We start in Section 4.1 by describing the broadcast group broker topology. Then, in Section 4.2, we describe the group formation process. In Section 4.3, we introduce our proof-of-concept broker prototype that extends an existing open-source pub/sub broker implementation. Finally, we discuss and summarize this contribution in Section 4.4 and Section 4.5.

This chapter is based on material previously published in the Proceedings of ICFC 2020 [82].

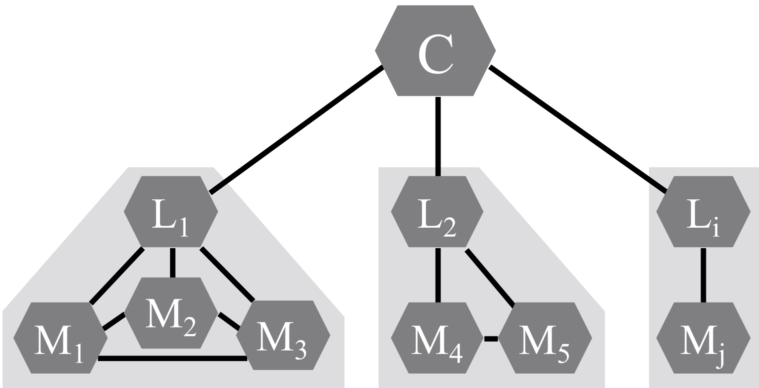


Figure 4.1: The broker topology comprises members (M), leaders (L) and the cloud RP (C).

4.1 Broker Topology and Message Dissemination

BCGroups builds upon a set of inter-connected brokers arranged in three tiers as shown in Figure 4.1. Each topology comprises the cloud RP C, a set of leaders $L_1, \ldots, L_i$, and a set of members $M_1, \ldots, M_j$. This topology is the result of the group formation process (Section 4.2) that assigns individual fog brokers[5] to a single leader. A leader and its members form a broadcast group (the gray area in the figure); each broadcast group must, at least, consist of a single leader.

Within a broadcast group, members and leaders act as regular pub/sub broker instances, i.e., they allow clients to connect, subscribe, unsubscribe, and publish events. Furthermore, the brokers within a group exchange (un-)subscribe (see Section 4.1.1) and publish event (see Section 4.1.2) messages. Leaders also exchange messages with the cloud RP for inter-group communication.

4.1.1 Dissemination of Subscriptions

Dissemination of subscriptions is based on three principles. We explain these three principles by using an example with three subscribers (S_1, S_2, S_3) and multiple brokers (Figure 4.2). Dissemination of unsubscribe messages is done in a similar way, so we refrain from describing this process separately. Note that besides subscription dissemination, brokers also store each subscription to match events; in the case of unsubscribe messages, they remove the corresponding subscription upon receiving it.

Members forward subscriptions to their leader. M_1 forwards the red[6] subscription created by S2 to its leader L_1. Similarly, M_2 forwards the red subscription created by S_3 to its leader L_1.

[5]Note, that any broker may, in fact, be a clustered pub/sub system.
[6]In the figures, we use colors to differentiate between topics.

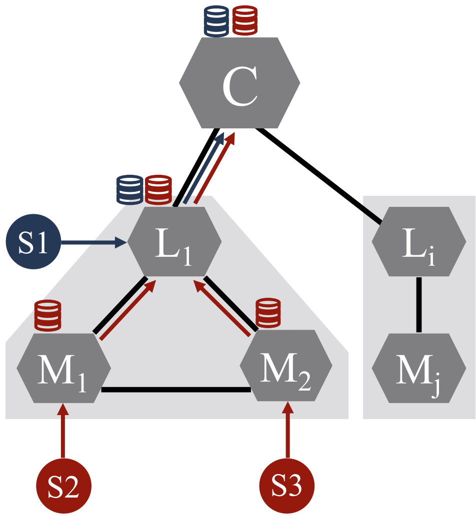


Figure 4.2: Dissemination of subscriptions is based on three principles.

Leaders merge subscriptions. M_1 and M_2 both forward a red subscription to their leader L_1. L_1 merges these two similar subscriptions into a single one, i.e., it only stores one red subscription.

Leaders create subscriptions at the cloud RP. L_1 stores a blue and a red subscription; it received the former from the local subscriber S_1, and the latter is the result of merging two subscriptions from members. Thus, L_1 creates a blue and a red subscription on behalf of the entire group at the cloud RP. This has the advantage of relieving (computationally) weaker group members from managing a connection to the cloud RP. It also prevents multiple subscriptions to the same topic by individual group members because the leader merges such subscriptions.

4.1.2 Dissemination of Events

Dissemination of events is based on four principles. We explain these four principles by using an example with two publishers (P_1, P_2) and multiple brokers (Figure 4.3). Note that brokers also deliver incoming events to local subscribers with a matching subscription; this is not shown in the figure.

Brokers broadcast events that they receive from clients to their group. M_1 broadcasts blue events from P1 to its group, i.e., to M_2 and L_1. This leads to excess data — M_2 does not store a blue subscription and therefore discards blue events. L_2 broadcasts red events from P2 to its group, i.e., to M_3 and M_4. Both brokers store a red subscription, so this does not lead to any excess data.

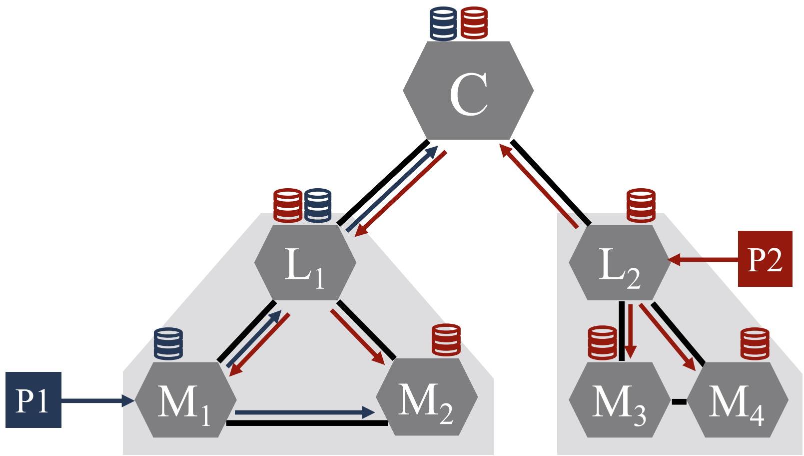


Figure 4.3: Dissemination of events is done via flooding and rendezvous points.

Leaders forward events from clients and their member brokers to the cloud RP. L_1 receives the blue event of P1 from M_1. L_1 forwards the event to the cloud RP. L_2 receives the red event of P2 directly. Still, L_2 forwards the event to the cloud RP.

The cloud RP distributes events to group leaders that have created a matching subscription. When receiving an event from a group leader, the cloud RP checks whether any other leader has created a matching subscription. In the example, this is not the case for blue events, as L2 has not created a blue subscription. For red events, however, L_1 and L_2 have created a matching subscription. Thus, the cloud RP forwards red events that it receives from L_2 to L_1.

Leaders broadcast events that they receive from the cloud RP to their group. When receiving red events from the cloud RP, L_1 broadcasts these events to its group, i.e., to M_1 and M_2. This leads to excess data — M_1 does not store a red subscription and therefore discards red events.

4.2 Group Formation Process and Leader Election

Initially, each broker joining the system takes the role of a leader and forms its own broadcast group. Leaders subscribe to a dedicated topic at the cloud RP. They also regularly publish their IP address to this topic to announce their presence to other leaders. In the case of very large deployments, leader announcements can be partitioned by using diverse topics. For example, brokers in Europe could subscribe and publish to the topic *leaders/europe* while brokers in North America could use the topic *leaders/northamerica*.

Leaders continuously monitor the latency to other known leaders, e.g., by sending ICMP echo requests and measuring the reply delay [135]. When a leader observes a latency below a threshold, it initiates a group merge (see Algorithm 1). Part of the group merge process is

Algorithm 1 Group merge, join, and notification.

```
function doGroupMerge(otherLeader)
    newLeader ← negotiateLeader(otherLeader)
    if newLeader = otherLeader then
        notifyMembersAboutMerge(newLeader)
        joinLeader(newLeader)
    end if
end function

function joinLeader(newLeader)
    connectToLeader(newLeader)
    brokers = getBrokersInBroadcastGroup(newLeader)
end function

function onMergeNotification(newLeader)
    if latencyBelowThreshold(newLeader) then
        joinLeader(newLeader)
    else
        createNewBroadcastGroup()
    end if
end function
```

the leader election, which can build on properties such as the compute power or the current bandwidth to the cloud RP. In many cases, however, it is sufficient to assign a value to each broker on startup that indicates available resources; we call this value *Leadership Capability Measure (LCM)*. During the negotiation, the leaders exchange their LCMs, and the leader with a higher LCM becomes the leader of the joint group.

Members continuously measure the latency to their leader. If a member observes a latency above a threshold, it leaves the broadcast group. By leaving the broadcast group, this broker automatically becomes the leader of a new broadcast group that only comprises a single broker. Members also start their own broadcast group when they receive a merge notification from their leader (during the merging process of two broadcast groups) but the latency to the new leader is above the latency threshold (see function `onMergeNotification` in Algorithm 1). One solution to avoid oscillating membership is to use two latency thresholds: a lower one for joining and a higher one for leaving. In addition, whether latency exceeds or falls below a threshold should also be based on a moving average of observed latency values.

The group formation process terminates when the following two conditions evaluate to true: (i) For every leader, the latency to all other leaders is above the latency threshold. (ii) For every member, the latency to its leader is below the latency threshold. If either condition is violated, e.g., because of infrastructure changes or broker failure, group formation continues until both conditions are met again.

It is possible to run BCGroups-compliant brokers alongside unmodified vanilla brokers: First, the cloud RP that ensures global communication can be any MQTT-compliant broker. The reason for this is that broadcast groups appear to be a single MQTT client as leaders essentially act as clients on behalf of their broadcast group's members. Second, other brokers can be vanilla brokers as well, as long as they can connect to the cloud RP as an MQTT client. Because vanilla brokers cannot join broadcast groups, other leaders simply assume that they are unwilling to join for latency reasons when running through the group formation process.

4.3 Proof-of-Concept Implementation

As a proof-of-concept, we extended the implementation of the popular MQTT broker Moquette [60] with the features necessary for BCGroups. Our implementation is available on GitHub[7]. While it is necessary to add some functionality for the formation of broadcast groups (see below), our *BCGroups-brokers* can also interoperate with unmodified vanilla brokers as explained in Section 4.2. For example, in the subsequent evaluation (see Chapter 8), we use a standard Eclipse Mosquitto [47] broker as cloud RP.

For our proof-of-concept prototype, we added the following functionality to Moquette:

- Leader announcements and continuous latency measurements to other leaders.
- Group merges in compliance with the group formation process.
- Latency-aware members, i.e., they leave a broadcast group when they measure a higher latency to their leader than allowed by the latency threshold.
- Broadcast group-specific dissemination of subscribe, unsubscribe, and event messages.
- Communication with a cloud broker (can be any MQTT compliant broker); for this, we use the Eclipse Paho [48] MQTT client.

4.4 Discussion

The group formation process (see Section 4.2) optimizes locally but therefore may not lead to a globally optimized topology. For example, two very well-connected brokers that exchange large amounts of data might end up in separate broadcast groups, and therefore have to communicate via the cloud. To create a globally optimized topology, a centralized service could create broadcast groups based on the analysis of message flows. Having such a single point of failure, however, which also has to retrieve meta-data from all broker instances, is not a good fit for a fog environment. We believe that making brokers more aware of their environment and message

[7] https://github.com/MoeweX/moquette

flows is a more promising direction for future work. For example, brokers could detect that most relevant messages originate at a broker from another broadcast group and update their membership accordingly.

Furthermore, during the group formation process, brokers currently use a global and fixed latency threshold. As future work, one could aim to let broadcast groups define their own threshold based on local conditions. Then, broadcast groups that can tolerate a higher amount of excess data could increase their threshold to improve latency, while overloaded broadcast groups can decrease their threshold to reduce excess data. Still, such changes to the group formation process must be carefully evaluated: Does the group formation process still lead to a stable result and do potential benefits outweigh increased overheads?

Regarding the dissemination of events, we chose flooding for intra-group communication because it offers the lowest communication latency and copes well with mobile subscribers; excess data remains manageable as long as broadcast groups do not become too large. However, depending on the infrastructure environment and workload, it is certainly possible to also use other strategies that, while not being an option when distributing messages across "the entire fog", are feasible for machines in the same broadcast group. Note, however, that other strategies often involve a *warmup phase*[8] or lead to higher communication latency. Depending on the use case, this might not be acceptable.

4.5 Summary

In this chapter, we presented BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems. BCGroups builds on broadcast groups to minimize the communication latency of client devices in physical proximity and reduce the dissemination of excess data for global communication.

In essence, the group formation process transitions from a pure centralized RP solution (every broker is its own leader) to an intra-group flooding solution. The latency threshold controls the size of resulting broadcast groups and thus the tradeoff between excess data dissemination and latency. By decreasing the threshold groups become smaller which reduces excess data dissemination but increases latency. Increasing the threshold has the opposite effect. In the presence of failures or changing network conditions, brokers can always fall back to connecting to the cloud RP. This ensures continuous, global message delivery as long as brokers can connect to the cloud, i.e., overall availability is at least as good as a pure cloud-based solution, but possibly higher through group internal message distribution in the fog. Even if a leader node fails, its member nodes will start their own, individual broadcast groups before running through the group formation process again. As this process does not require central orchestration, network

[8] During the warmup, subscriptions to a topic formerly unknown to the broker system cannot be served immediately as the subscriptions must first be distributed to all broker instances.

partitions also do not prevent group formation in general; only the broadcast groups that cannot communicate with the cloud anymore are affected. Furthermore, local traffic between clients connected to the same broker is always delivered, even if there is temporarily no connection to any other broker available.

Chapter 5

GeoBroker: Leveraging Geo-Context for IoT Data Distribution

In this chapter, we describe the design of GeoBroker, a pub/sub broker system that leverages the full geo-context of publishers and subscribers to reduce excess data dissemination and facilitate the development of new, pervasive IoT applications. While this kind of domain knowledge is not available in every IoT scenario, using it significantly improves data distribution efficiency when it exists. GeoBroker offers similar functionality and operational behavior as other pub/sub systems, while adding the capabilities necessary to use the geo-context dimensions that we introduced in Section 2.4.

To ease integration into real systems beyond research prototypes, we use the same message types as MQTT v5.0 (see Section 2.3.2), but piggyback geo-context information on top of them. Therefore, GeoBroker uses a topic-based ContentCheck. Still, one could also add a ContentCheck with another expressiveness level, such as a content-based ContentCheck.

We start in Section 5.1 by describing GeoBroker's basic functionality. Then, in Section 5.2, we describe GeoBroker's event matching process that comprises a ContentCheck and two additional GeoChecks. With the event GeoCheck, publishers can control which subscribers receive messages based on the event geofence and subscriber location. With the subscription GeoCheck, subscribers can control from which publishers they receive messages based on the subscription geofence and publisher location. To do this efficiently, we designed a data structure that indexes subscriptions; we describe this structure and how it handles subscription updates in Section 5.3. In Section 5.4, we introduce our proof-of-concept implementation. Finally, we discuss and summarize this contribution in Section 5.5 and Section 5.6.

This chapter is based on material previously published in the Proceedings of ISYCC 2019 [76], in the Computer Communications journal [75], and the Software Impacts journal [74].

5.1 GeoBroker Functionality

As in a typical MQTT system, clients first connect to GeoBroker and create a session. These sessions expire after a specific time so that clients periodically send keep-alive (ping) messages to GeoBroker. In “connect” and “ping” messages, clients also include their current location, which is stored by GeoBroker; this applies to clients no matter whether they are subscribers or publishers. GeoBroker acknowledges connect, ping, and the other messages types introduced below to detect lost messages.

Comparably to regular MQTT clients, GeoBroker clients can act as publishers and subscribers at the same time. Subscribers with an active session can create and delete subscriptions. In GeoBroker, however, they can also update the subscription geofence of each subscription. Whenever a publisher publishes an event, GeoBroker matches the topic and geo-context information of the event with the information of its managed subscriptions and then distributes the event accordingly to subscribers.

Sessions terminate when the respective client sends a disconnect message or after a time-out. Terminating a client session also suspends all active subscriptions.

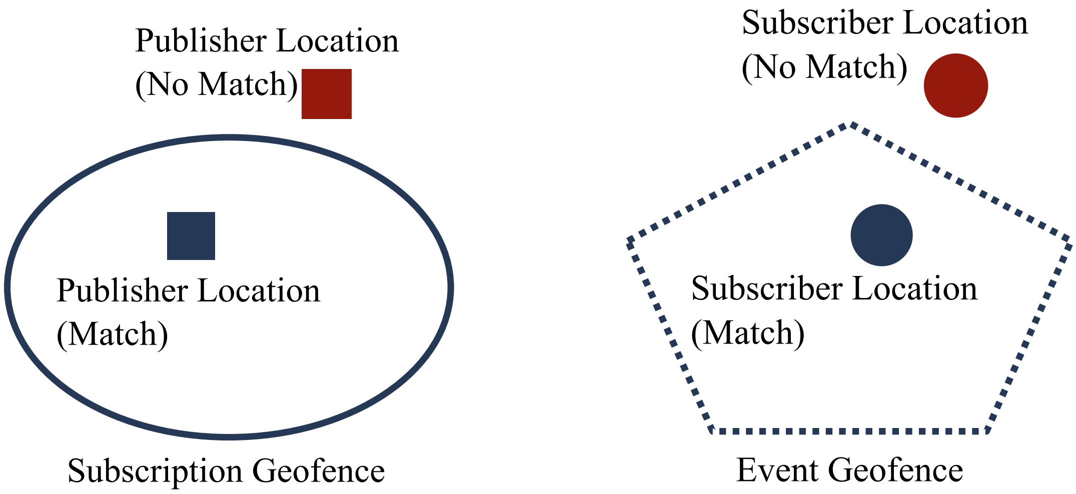
An essential characteristic of broker-based pub/sub systems is that clients are completely decoupled: Publishers can publish events to GeoBroker without worrying whether any subscribers are connected, ready to receive events, or crashed as this is handled entirely by GeoBroker.

5.2 Event Matching

GeoBroker extends standard event matching and also considers geo-context information rather than content information (in the form of topics) only. For each published event that GeoBroker receives, GeoBroker runs the following checks to determine to which subscribers the event should be delivered.

1. ContentCheck: checks whether the subscription topic matches the event topic based on MQTT principles (see Section 2.3.2).
2. Subscription GeoCheck: checks whether the subscription geofence contains the publisher location (see Section 5.2.1).
3. Event GeoCheck: checks whether the event geofence contains the subscriber location (see Section 5.2.2).

GeoBroker only delivers a given event to subscribers that have passed all three checks. We explicitly decided on this order as the ContentCheck requires less computation than the two GeoChecks. In corner cases, where clients are spread across a large area, however, running



(a) Subscription GeoCheck

(b) Event GeoCheck

Figure 5.1: Each GeoCheck uses two of our four geo-context dimensions.

the GeoChecks first might be more efficient. Regarding the ordering of subscription and event GeoCheck, we run the subscription GeoCheck first as subscription geofences can be efficiently stored and indexed in advance, whereas the event geofence is not known until the respective event arrives at GeoBroker.

In scenarios where a publisher or subscriber might wish not to limit data distribution, one or both GeoChecks can be omitted. Depending on the pub/sub system's implementation, this can be achieved by either supplying no geofence (which indicates to skip the corresponding GeoCheck) or a geofence that comprises all clients (the corresponding GeoCheck will be successful for any client location).

5.2.1 Subscription GeoCheck

With the subscription GeoCheck (see Figure 5.1a), a subscriber can limit data distribution if it already knows that only data from a specific area is relevant and wishes to avoid being overloaded by excess data. The subscription GeoCheck is only run on subscriptions that passed the ContentCheck. An efficient spatial indexing structure should carry out the check. When a subscriber creates or updates a subscription, the subscription geofence (for the given topic) has to be stored in the indexing structure. For the subscription GeoCheck with a given topic, the subscription indexing structure must return all subscriptions with a subscription geofence that contains the publisher location already known to GeoBroker[9]. We describe the design of our spatial indexing structure in Section 5.3.

[9]As explained in Section 5.1, publishers set and update their location with connect and ping messages.

5.2.2 Event GeoCheck

With the event GeoCheck (see Figure 5.1b), a publisher can limit data distribution for which it can have two motivations: First, as a form of access control, if the location can be sufficiently trusted[10], e.g., only visitors of a building should get access to its smart home data. Second, to use domain knowledge that is not available to the subscribers. For example, if a publisher sends out wireless emergency alerts [51], only the publisher knows which area is affected by the corresponding event.

The event GeoCheck is only run on subscriptions that passed the ContentCheck and the subscription GeoCheck. In this final step, GeoBroker checks whether the event geofence contains the corresponding subscriber locations. If so, it delivers the event to the respective subscribers.

5.3 Subscription Indexing Structure

All information necessary for the ContentCheck and subscription GeoCheck is available before GeoBroker processes an event. Thus, GeoBroker can store subscription-related information in a data indexing structure for efficient retrieval. Note that the event geofence is part of a published event and is thus not available beforehand, therefore an indexing structure cannot support the event GeoCheck.

Approaches for spatial-keyword matching already exist today, e.g., Wang et al. proposed the AP-Tree [184] and showed that it is more efficient than other solutions. With spatial-keyword matching, however, ContentCheck and subscription GeoCheck information are stored in the same data structure, so it is non-trivial/challenging to change the type from topic-based to content-based and vice versa. Therefore, we designed our own subscription indexing structure that

- is capable of first running the ContentCheck before doing the subscription GeoCheck,
- has a low updating overhead as subscribers might be mobile and use subscription geofences that move with them,
- supports multi-threading.

Our main idea is to use a standard indexing structure for the ContentCheck and embed a second data structure to efficiently run the subscription GeoCheck.

In the case of GeoBroker, the ContentCheck is done based on MQTT topics. Popular MQTT brokers such as mosquitto [47] or moquette [60] use a directed rooted tree to efficiently match topics. Therefore, we use a similar **topic tree** structure for the ContentCheck which stores

[10] Some solutions to build such trust already exist [54], but they still have to prove their practical usability.

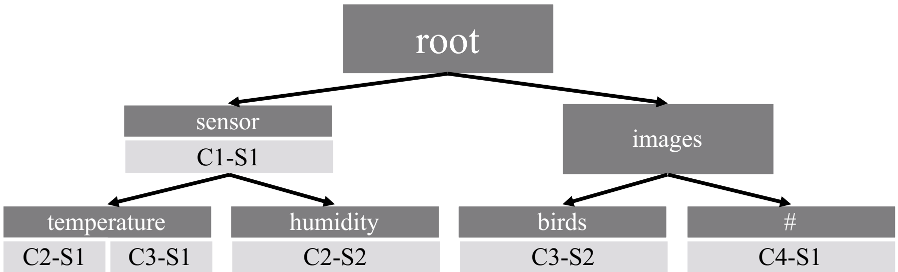


Figure 5.2: Topics are stored in a directed rooted tree.

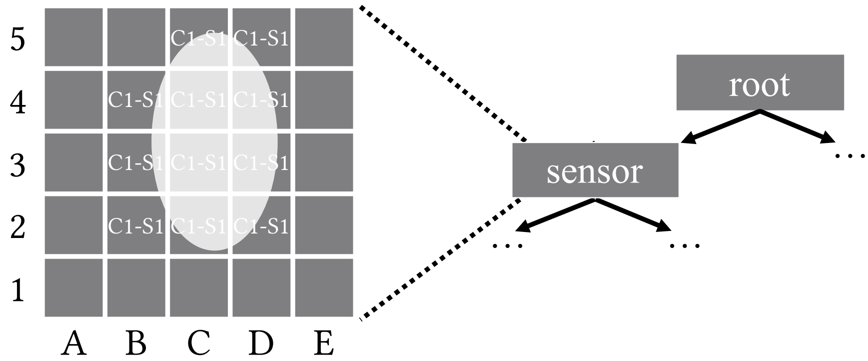


Figure 5.3: Nodes in the topic tree contain a raster as embedded spatial-indexing data structure.

topic levels in tree nodes. Figure 5.2 shows an example of such a topic tree in which clients (C1 – C4) have created various subscriptions (Si, i being the index of the client's subscriptions) for specific topics (e.g., *sensor/temperature*) and a wildcard topic (*images/#*)[11]. When GeoBroker receives a published event, it traverses the tree until it finds the node that stores the matching topic and thus all matching subscriptions. Note that with wildcards, it is possible to find multiple nodes.

Each tree node of the topic tree contains a **raster** as an embedded spatial-indexing data structure (see also Figure 5.3) that allows GeoBroker to efficiently identify the subscriptions that contain a given publisher location.

A raster stores all corresponding subscriptions in a 2D data structure that divides the available geographic space (e.g., the surface of the earth) into rectangular areas (**raster fields**). Raster fields can be uniquely identified and accessed via the coordinate of their respective southwest corner. Furthermore, raster fields do not overlap, directly border each other, and exactly one has its southwest corner at the point of origin (0°/0°).

Each raster field contains a list of all subscriptions that have an intersecting subscription geofence; see Figure 5.3 for an example showing a subscription created by client C1 that targets the topic *sensor* and has an almost circular geofence. The idea behind this is to reduce the number of "contains" checks that are required to identify which subscription geofences con-

[11]More information on topic filters can be found in Section 2.3.2

tain a given publisher location as this check becomes increasingly compute-intensive with more complex shapes. With the raster, only the subscriptions that are referenced in the same raster field as the one containing the publisher location need to be checked. Furthermore, if a specific subscription geofence covers a complete raster field, e.g., C4 in Figure 5.3, the check can be omitted entirely since it is certain that the subscription geofence contains the given publisher location.

Smaller raster fields mean that fewer subscription geofences need to be checked. However, this makes subscription updates more costly and increases the overall size of the index data structure as each subscription reference needs to be added to/removed from more raster fields. For this obvious tradeoff, the raster field size provides the tuning knob to balance event matching and subscription update costs. In addition, the "optimal" raster field size also depends on the average geofence size and shape. To control the raster field resolution, we added a parameter called granularity. We then defined the side length of each raster field as *1°/granularity*[12], so when a user increases the granularity, the raster fields become smaller.

Instead of our own raster approach, we could have used another spatial indexing structure such as an R-Tree [70, 107] or a B-Tree, which stores spatial regions encoded as bit strings [173]. We tried both approaches but faced performance issues before coming up with our raster-based design. We did an experiment[13] with 100k geofence operations (25k adds, 25k updates, 50k gets) executed in a single-thread. The R-Tree implementation needed about 72 seconds; most time was spent traversing the tree due to the necessary bounding box checks. The B-Tree implementation needed about 232 seconds. Here, most time was spent on identifying potential keys and removing false-positive subscriptions. For the same setup, our raster-based solution completed all operations in about 1.6 seconds.

At this point, we would like to emphasize that optimizing the subscription indexing structure is not the focus of this thesis. However, as indicated above, initial micro-benchmark results show that our proposed indexing structure has a high performance. Our experiments also support this claim (see Chapter 9) as they show that the overhead of doing the GeoChecks remains manageable. Future work could, however, further explore and compare different data structures to identify the best possible solution.

Updating Subscriptions

Each time a client subscribes or unsubscribes, the topic tree and raster need to be updated. In the following, we discuss how this is done when a new subscription is added as the steps for updating or removing a subscription are almost identical.

[12] We chose degree rather than meter as measurement unit since the earth has a spherical surface and our raster fields remain quasi-rectangular. However, choosing degree has the downside of raster fields becoming smaller when moving away from the equator in terms of their real size when measured in meters.

[13] The prototype for this experiment was implemented by one of our students as part of his Bachelor's thesis [65].

Algorithm 2 Updating subscriptions: identify raster fields that intersect with the area inside a subscription geofence bounding box.

```
function CALCULATEKEY(location)
    lat = floor(location.lat * granularity) / granularity
    lon = floor(location.lon * granularity) / granularity
    return (lat/lon)
end function

function MAIN
    swInd = calculateKey(southWestBoundingBoxCorner)
    neInd = calculateKey(northEastBoundingBoxCorner)
    for lat = swInd.lat To neInd.lat Step 1 / granularity do
        for lon = swInd.lon To neInd.lon Step 1 / granularity do
            results.add(raster field with key (lat/lon))
        end for
    end for
    return results
end function
```

To create a new subscription, GeoBroker first traverses the topic tree until it finds the node which corresponds to the subscription topic. Then, GeoBroker determines the raster fields that intersect with the subscription geofence as the subscription has to be added to these fields. It is compute-intensive to identify these out of all the raster fields stored in the raster, so GeoBroker only checks the raster fields that intersect with the area inside the geofence's outer bounding box, which is trivial to calculate. Identifying such fields is inexpensive when done with Algorithm 2 using the raster field keys[14]. For efficient removal of old subscriptions, GeoBroker can either cache the subscription geofences or clients can provide the old geofence as part of their request.

As an example, consider Figure 5.4 in which the raster fields B2 to B5, C2 to C5, and D2 to D5 are intersecting with the outer bounding box. Note that B5 is a false positive in this case, as it intersects with the outer bounding box but not with the geofence itself. Thus, in a second step, it is necessary to additionally check each identified raster field for intersection with the geofence.

In theory, it is possible to omit this final intersection check, as false positives caused by wrongly added subscriptions to raster fields are also removed when retrieving subscriptions during the subscription GeoCheck (see Section 5.2.1). This, however, only makes sense if a workload is very update-heavy and shapes have circular or rectangular patterns as otherwise too many false positives need to be removed for each event.

[14] The computational effort scales linearly with the number of raster fields inside the bounding box.

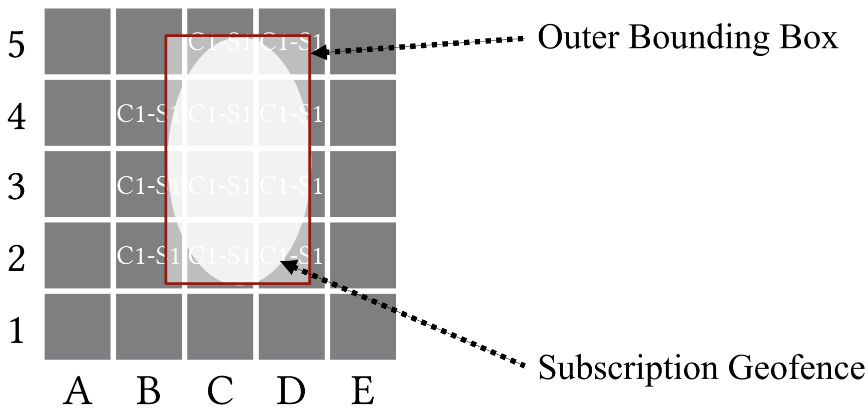


Figure 5.4: It is only necessary to check the raster fields inside the geofence's outer bounding box for intersection with the geofence when updating subscriptions.

5.4 Proof-of-Concept Implementation

As a proof-of-concept, we implemented the pub/sub broker GeoBroker[15] and a client in Java 8 and Kotlin with the functionality described in Section 5.1.

For the communication between GeoBroker and clients, we use the Java version of ZeroMQ[16]. ZeroMQ is a networking library that builds on top of a high-speed asynchronous I/O engine [89, p. xiii ff.]. Its sockets can communicate in-process, inter-process, via TCP, and multicast, so it is not just a networking library but can also be used as a concurrency framework. As ZeroMQ manages connections, a single socket can be used to handle thousands of clients.

In contrast to vanilla MQTT messages, which are encoded as defined by the MQTT v5.0 protocol, we serialize messages with Kotlin directly [94] before handing them over to ZeroMQ. So while our message types are similar to the ones of MQTT (e.g., we have a CONNECT message to establish a connection between clients and GeoBroker and a CONNACK message to acknowledge a connection), the messages themselves look different. We chose this approach as it allows us to easily enhance the messages with additional information while also not forcing us to implement all MQTT messaging features. For example, the PINREQ message, which clients use to reset their session timers, does not support carrying a payload originally; for our approach, however, we use the ping functionality to update client locations, so appending a payload to this type of message is necessary.

As ZeroMQ can be used as a concurrency framework, we also use ZeroMQ for GeoBroker's internal communication between threads for scalability. Figure 5.5 shows a simplified version of the GeoBroker architecture. Clients use a ZeroMQ dealer socket to connect to the ZeroMQ router socket of the GeoBroker communication manager (this provides asynchronous communication between both parties [89, p. 88]). Internally, GeoBroker uses an arbitrary number of

[15] `https://github.com/MoeweX/geobroker`
[16] https://github.com/zeromq/jeromq

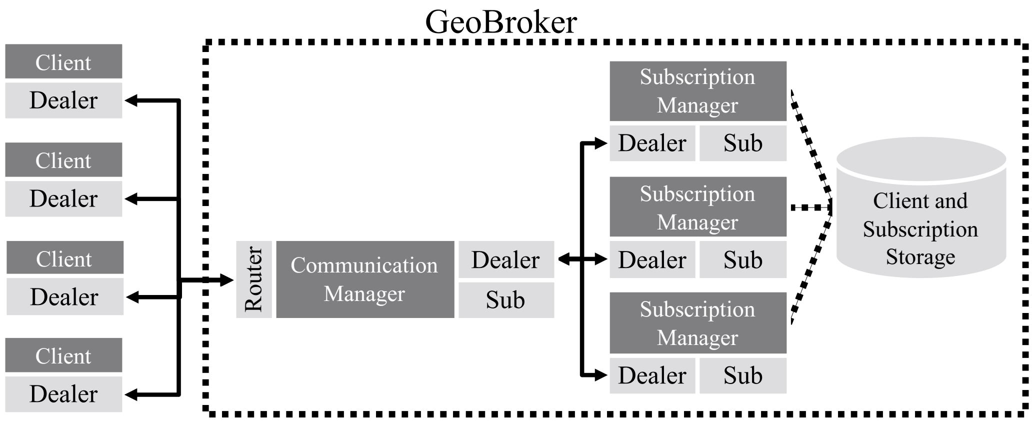


Figure 5.5: GeoBroker uses ZeroMQ for internal and external communication.

subscription managers, which each runs in a separate thread. These subscription managers use the client and subscription storage to manage connected publishers and subscribers as well as to match events with active subscriptions. As the storage implements our subscription indexing structures for content and geo-context information (see Section 5.3), it can efficiently retrieve the information needed for the three event matching checks. The subscription managers use a dealer socket to connect to the dealer socket of the communication manager; this socket type combination gives us asynchronous communication between both parties as well; however, when the communication manager signals that a publisher's event is available for processing, only a single subscription manager will receive it. The communication manager itself has virtually no load as it only forwards messages to the subscription managers.

GeoBrokers internal components also use a Sub socket to receive broadcasted instructions such as "shut down"[17]. In the figure, we exclude the broadcasting components to improve readability.

5.5 Discussion

With the event geofence, publishers can limit data access without being aware of the actual subscribers. This can be a very useful feature in many situations. For example, smart buildings can continuously publish their data to the same topic using a geofence that represents the building's shape to ensure that no one from the outside receives anything without having to worry about updating access control lists. This, however, requires trust in the location provided by a data consumer. While some solutions for that already exist [54], these still have to prove their practical usability.

Doing GeoChecks as part of the event matching process is also not another form of content-based pub/sub. In contrast to content-based filtering, our system also allows publishers to define delivery criteria, i.e., an event might be filtered/not delivered to a subscriber based on

[17] To do that via such a Sub socket is recommended [89, p. 57f.].

restrictions put into place by the publisher rather than the subscriber only. Furthermore, the geo-context of a client is not necessarily related to the content it receives/distributes. For instance, a monitoring service that publishes a warning might be running at a different location than the monitored systems. Separating content and geo-context information also has the advantage of being payload-agnostic.

We also want to emphasize that event and subscription geofences can have arbitrary shapes. However, if the shape gets more complex, the required "contain" operations become more computationally intensive which increases the get and update latency, as well as the CPU load of GeoBroker. Nevertheless, as the clients do not carry out these checks, this does not affect their performance. Thus, the approach is still well suited for clients operating in constrained environments, but more broker resources might be required. Our raster approach based on bounding boxes can alleviate parts of that extra complexity.

Beyond this, our index data structure has been proven useful for other purposes: it is part of the data analysis pipeline of the SimRa project [99] where near-miss incidents in bicycle traffic – recorded with the usual GPS accuracy – need to be mapped to the corresponding street segment or intersections. As part of their Master's thesis, two of our students also confirmed that GeoBroker's matching process can, with minor adaptions, scale horizontally to multiple machines arranged in a cluster. While the first [166] student used a load balancer to connect multiple GeoBroker servers, the second student [31] combined horizontal data partitioning and range partitioning for a GeoBroker system prototype that runs on top of Kubernetes[18].

5.6 Summary

In this chapter, we presented GeoBroker, a pub/sub broker system that leverages geo-context for IoT data distribution. GeoBroker offers similar functionality and operational behavior as popular pub/sub broker systems already used today. As such, it is a general solution that can be used by various applications for different purposes simultaneously. We added, however, an an efficient subscription indexing structure that also stores geo-context information and an extended event/subscription matching process: Besides the normal *ContentCheck*, e.g., does the topic of a published event match the topic of a subscription, our matching process has two additional *GeoChecks*. With the event GeoCheck, publishers can control which subscribers receive messages based on the event geofence and subscriber location. With the subscription GeoCheck, subscribers can control from which publishers they receive messages based on the subscription geofence and publisher location.

[18]Kubernetes is a production-grade container orchestration system [4].

Chapter 6

DisGB: Leveraging Geo-Context for Inter-Broker Routing

In this chapter, we describe the design of DisGB, a distributed pub/sub broker system that leverages geo-context for efficient inter-broker routing. Current state of the art solutions, e.g., [134, 147], distribute RPs (see Section 2.3.3) uniformly over available brokers which works well if the message traffic is also uniformly distributed. IoT data traffic, however, is often non-uniformly distributed: published events are most relevant to devices located in a particular geographical area. This relevance can be expressed with geo-context information, as discussed in Section 2.4. DisGB builds upon the GeoBroker system (Chapter 5): besides the event GeoCheck and subscription GeoCheck, DisGB additionally uses geo-context information to select RPs that are either close to the publisher or subscribers of an event.

We start in Section 6.1 with the discussion of assumptions before we describe how geo-context information can be used to select RPs close to subscribers (Section 6.2) or close to publishers (Section 6.3). Selecting RPs close to clients minimizes end-to-end latency (see Section 2.3.3), both strategies come with their own advantages and disadvantages. Which one is better depends on the application scenario; we discuss this in Section 6.4. In Section 6.5, we introduce our proof-of-concept implementation. Finally, we discuss and summarize this contribution in Section 6.6 and Section 6.7.

This chapter is based on material previously published in the Proceedings of ISYCC 2019 [76], FGFC 2020 [77], and UCC 2020 [73].

6.1 Assumptions

For our approach, we assume a setup that comprises multiple geo-distributed brokers and clients, i.e., IoT devices and applications. Even though brokers are geo-distributed, they are

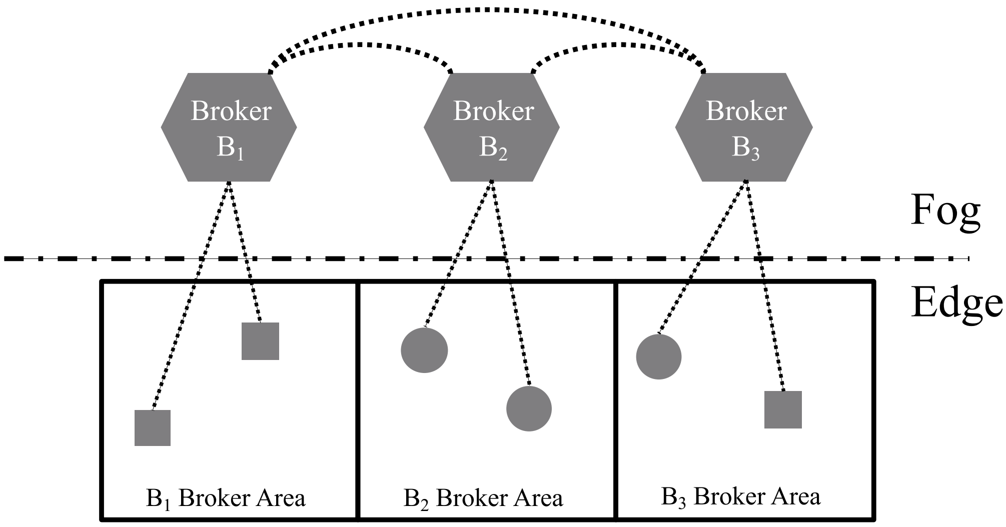


Figure 6.1: Setup with three brokers that support communication between publishers (squares) and subscribers (circles).

aware of each other, typically have a good inter-connection, and are well equipped in terms of computing power. On the other hand, clients might operate in a constrained environment and only communicate with the physically closest broker: their LB (local broker, see Section 2.3.3). Consequently, a broker is responsible for communication with all clients located in the region surrounding its physical location as this asserts low communication latency[19]. We refer to this region as *broker area*, see also Figure 6.1.

Subscriptions and published events comprise a payload, a filter for the ContentCheck (e.g., a topic), and geo-context information. When a client creates a subscription, it creates the subscription at its LB. Similarly, when a client publishes an event, it sends the event to its LB. Depending on the strategy (Sections 6.2 and 6.3), as soon as the LB has received an event or subscription, it distributes them to the RPs where the matching occurs.

6.2 Selecting RPs Close to the Subscribers

With this strategy, the RPs for an event are all brokers that are the respectively closest broker to each of the subscribers that have created a matching subscription. Thus, the RPs are the LBs of these subscribers. Hence, subscriptions are not distributed to other brokers as subscribers create subscriptions at their LB. The event, on the other hand, is distributed to all brokers that might manage a matching subscription. Fortunately, in contrast to event flooding, the event geofence can be used to select these RPs, because only broker areas intersecting with the event geofence can contain subscribers that pass the event GeoCheck (subscriber location inside event geofence).

[19]Using the network distance instead of physical distance to determine local brokers might be more accurate, but is also more complicated in an environment with changing network conditions. Studies have also shown that often both lead to similar results [90, 170].

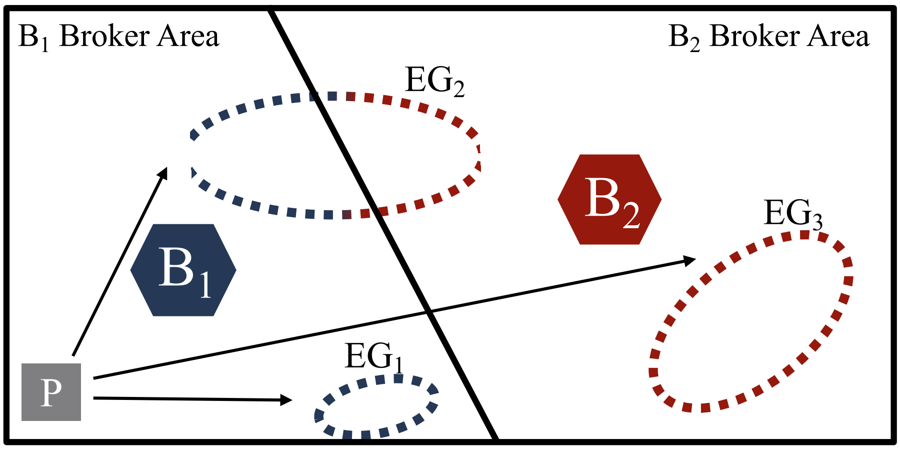


Figure 6.2: An event only needs to be forwarded to brokers with a broker area that intersects with the event geofence.

Figure 6.2 shows an example with one publisher (P) that is located in the broker area of broker B_1. P publishes three events, each event has a different event geofence (EG):

- EG_1 does not intersect with the broker area of broker B_2 (on the right) so the event does not need to be forwarded by B_1 for matching to B_2.
- EG_2 intersects with the broker areas of B_1 and B_2 so the event needs to be matched at B_1 <u>and</u> must be forwarded by B_1 for matching to B_2.
- EG_3 only intersects with the broker area of B_2, so the event must be forwarded by B_1 for matching to B_2. Note that matching at B_1 can be omitted, as no subscription created by the subscribers located in the broker area of B1 can pass the event GeoCheck.

This strategy's key benefit over state-of-the-art strategies is that by using geo-context information, events are sent directly to only a small subset of brokers while subscriptions do not have to be distributed at all.

6.3 Selecting RPs Close to the Publishers

With this strategy, the RP for an event is the broker closest to the publisher of that event, i.e., the RP is the publisher's LB. Thus, matching only occurs at a single broker. In exchange, all subscriptions must be distributed to all brokers to which a matching event might be published; subscription updates must also be propagated similarly. Fortunately, in contrast to subscription flooding, the subscription geofence can be used to select these RPs, because only broker areas intersecting with the subscription geofence might contain publishers that pass the subscription GeoCheck (publisher location inside subscription geofence).

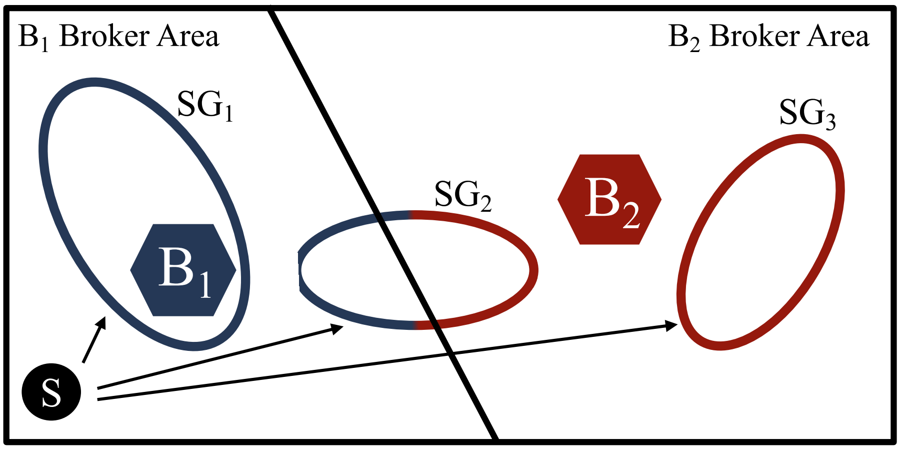


Figure 6.3: A subscription only needs to be forwarded to brokers with a broker area that intersects with the subscription geofence.

Figure 6.3 shows an example with one subscriber (S) that is located in the broker area of B_1. S creates three subscriptions, each subscription has a different subscription geofence (SG):

- SG_1 and the broker area of B_2 do not intersect, so the subscription does not need to be forwarded by B_1 to B_2.
- SG_2 intersects with the broker areas of B_1 and B_2 so the subscription needs to be maintained at B_1 and the subscription must be forwarded by B_1 to B_2.
- SG_3 only intersects with the broker area of B_2, so the subscription must be forwarded by B_1 to B_2. Note that the subscription can be discarded at B_1, as none of the publishers managed by B_1 can publish an event that passes the subscription GeoCheck.

After matching the event, the RP has to distribute the event (including the identity of the matching subscribers) to the LBs of matching subscribers as these brokers are the ones responsible for the final delivery. Still, in contrast to selecting RPs close to the subscribers, events are only distributed based on actual matches rather than on potential matches. Hence, the key benefit of this strategy over state-of-the-art strategies is that by using geo-context information subscriptions only need to be forwarded to a small subset of brokers and events are only forwarded to brokers that are confirmed to be the LB of a matching subscriber.

6.4 Scenario Analysis

Depending on the application scenario, each RP selection strategy results in a different number of RPs; the lower the number of well-chosen RPs, the lower the total number of messages. In this section, we first calculate how the two selection strategies affect the number of RPs (Section 6.4.1). Then, based on these calculations, we discuss which RP strategy is the best choice for three example scenarios (Section 6.4.2).

6.4.1 Calculating the Number of RPs

There are four client actions that require RPs:

i) a subscriber updates its location,

ii) a subscriber creates/updates[20]/deletes a subscription,

iii) a publisher updates its location,

iv) and a publisher publishes an event.

For this analysis, we define the following: When an event is published, E is the set of all broker areas that intersect with the event's geofence. Every subscriber has a set of active subscriptions $\{s_i \mid i \in \{1, \ldots, n\}\}$, with n being the subscriber's total number of subscriptions. S_i is the set of all broker areas that intersect with the subscription geofence of s_i. Finally, b is the total number of brokers.

i) Subscriber Location Update: An updated subscriber location must be distributed to all brokers that require it for the matching of events. When selecting RPs close to the subscribers, events are only matched at the LB of the subscriber, so the number of RPs is 1. When selecting RPs close to the publishers, events might be matched at any broker whose broker area intersects with any of the subscribers' subscription geofences, so the number of RPs is $|\bigcup_{i=1}^{n} S_i|$.

ii) Subscription Update: An update of subscription s_i must be distributed to all brokers that require it to match events. When selecting RPs close to the subscribers, events are only matched at the LB of the respective subscriber, so the number of RPs is 1. When selecting RPs close to the publishers, events might be matched at any broker whose broker area intersects with the geofence of s_i, so the number of RPs is $|S_i|$.

iii) Publisher Location Update: An updated publisher location must be distributed to all brokers that require it for the matching of events. When selecting RPs close to the subscribers, events might be matched anywhere, so the number of RPs is b. However, an LB could also piggyback the current publisher location on each event of the same publisher it has to distribute (see iv) below). In this case, the number of RPs is 1 for publisher location updates as they are not distributed separately. When selecting RPs close to the publishers, events are only matched at the LB of the publisher, so the number of RPs is 1.

iv) Event Publishing: A published event must be distributed to all brokers that match the event with managed subscriptions. When selecting RPs close to the subscribers, events are matched at any broker whose broker area intersects with the given event's geofence, so the number of RPs is $|E|$. When selecting RPs close to the publishers, events are only matched at the publisher's LB, so the number of RPs is 1.

[20] For example, to use another subscription geofence.

Table 6.1: Number of RPs for each type of client action and RP selection strategy.

Client Action Type	RPs at subscriber	RPs at publisher
Subscriber Location Update	1	$\lvert \bigcup_{i=1}^{n} S_i \rvert$
Subscription Update	1	$\lvert S_i \rvert$
Publisher Location Update	1	1
Event Publishing	$\lvert E \rvert$	1

In summary (Table 6.1), the number of RPs, and thus the overhead of inter-broker communication, depends on the scenario-specific workload. Selecting RPs close to the subscribers is better for workloads that involve

- many subscriber location updates,
- many subscription updates,
- and large subscription geofences that intersect with many broker areas

as subscription information does not need to be distributed by the LB. Selecting RPs close to the publishers, on the other hand, is better for workloads that involve

- a high volume of published events
- as well as large event geofences that intersect with many broker areas

as the events can be matched at the LB of each publisher.

6.4.2 Discussion based on Example Scenarios

In [76], we presented three (IoT) scenarios that use geo-context information. In the following, we again summarize these scenarios and then use them to discuss which RP selection strategy is better suited in what situation.

Open Environmental Data

In this scenario, IoT sensors provide data access to all clients that subscribe to related topics such as temperature, humidity, or barometric pressure. Clients subscribe to topics based on their individual content interests. Furthermore, by using a subscription geofence, they only receive data from sensors located in the specified geofence. For example, a smart blinds control system located in Delft could subscribe to the temperature topic and use a subscription geofence containing the Netherlands.

The most important geo-context related characteristics in this scenario are:

- Event geofences do not exist, so no event GeoCheck is needed.
- Subscription geofences have arbitrary size, as subscribers can be interested in very small, but also very large regions.
- Subscriptions are updated rarely as subscribers do not have to update their subscription geofences multiple times a day.
- Events (sensor readings) are published frequently, but there are no matching subscribers in many cases.

As event geofences do not exist, $|E|$ equals the number of all available brokers. Subscription geofences can have arbitrary sizes, so $|S_i|$ is somewhere between 1 and the number of all available brokers. This is an unfavorable combination as subscription updates as well as published events require inter-broker communication. The subscription update frequency, however, is lower than the event publishing frequency. Thus, selecting RPs close to the publishers is the better strategy. Another benefit of this strategy is that events are not distributed to brokers without a matching subscriber.

Local Messaging and Information Sharing (Hiking)

In this scenario, clients consume and share data from other clients in proximity while being mobile. For this, each client creates subscriptions to topics of interest and a subscription geofence that covers the immediate surrounding area. Furthermore, clients also publish events to fitting topics with an event geofence covering the immediate surrounding area.

The most important geo-context related characteristics in this scenario are:

- Geofences are small and unlikely to intersect with multiple broker areas.
- Publisher and subscriber locations are updated frequently as both are mobile.
- Subscriptions are updated frequently as subscription geofences depend on the respective subscriber location.
- Events are published frequently.

As both geofences only intersect with a tiny number of broker areas (often only with a single one), $|E|$ and $|S_i|$ usually equal 1. In addition, the publisher and all matching subscribers for a given event are likely connected to the same LB, so only a small amount of data has to be distributed to other brokers. Therefore, none of the two RP selection strategies has a clear advantage over the other one.

Context-based Data Distribution

In this scenario, events are delivered based on the content interests of subscribers and the domain knowledge of publishers. This way, all subscribers must only specify once what kind of data content they want to receive while publishers define in which geographic area their events are relevant. For example, citizens could subscribe to events that carry emergency alerts while public authorities can accurately define in which area their alerts should be received. Then, citizens could travel between districts or cities without having to update their subscriptions again and still get all relevant alerts. The most important geo-context related characteristics in this scenario are:

- Event geofences have arbitrary size but are considered to be relatively small and intersect with only a few broker areas.
- Subscription geofences do not exist, so no subscription GeoCheck is needed.
- Subscriber locations are updated frequently as subscribers are mobile.
- Subscriptions are updated rarely as subscribers have to subscribe to the desired content topics only once.
- Events can be published at any frequency as this depends on the kind of data (e.g., advertisements vs. emergency alerts).

As event geofences are considered to be small, $|E|$ usually equals 1. Subscription geofences do not exist, so $|S_i|$ equals the number of all available brokers. Subscriber location updates must be forwarded to $|\bigcup_{i=1}^{n} S_i|$ brokers, so due to the high update frequency, selecting RPs close to the publishers is an unfavorable strategy for this scenario. Instead, selecting RPs close to the subscribers is the better strategy as publisher locations do not need to be forwarded (no subscription GeoCheck) and $|E|$ is considerable smaller than $|\bigcup_{i=1}^{n} S_i|$ and $|S_i|$.

6.5 Proof-of-Concept Implementation

We have extended the GeoBroker prototype (Section 5.4) with the capability of using RPs for inter-broker communication. The source code of this distributed GeoBroker (DisGB) is available as open-source on GitHub[21]. When starting DisGB, one can select one of three modes: single, RP_subscriber, and RP_publisher. At single, DisGB behaves like GeoBroker, i.e., there is only one broker node to which all clients connect and thus no inter-broker communication. In the two other modes, DisGB supports a distributed broker deployment in which RPs are either selected close to the subscribers (RP_subscriber) or the publishers (RP_publisher). For the current prototype, broker addresses and non-overlapping broker areas must be provided as configuration data; more advanced solutions based on, e.g., dynamic discovery, could be added if needed.

[21] https://github.com/MoeweX/geobroker

To assert the correctness of our implementation, we ran a very simple workload with DisGB configured in single mode, RP_subscriber mode, and RP_publisher mode. In this workload, clients never update locations or subscriptions while others are publishing events, as this could lead to slightly different results depending on the RP selection strategy. For example, when selecting RPs close to the publisher, a location update from a subscriber might not have been received by a broker when matching a particular event. To mitigate this issues, all clients simultaneously either update locations, update subscriptions, or publish events; furthermore, after each type of action, they do nothing for five seconds to ensure no more messages are on the wire. We then verified that all messages had been sent in each run and that each client had received the same set of messages. We also verified that event matching is done correctly through another small experiment. Here, we determined for each published event the correct subscribers manually to then verify that the event matching carried out by our prototype delivers the same result.

6.6 Discussion

DisGB builds upon GeoBroker (Chapter 5) which means that the points discussed in Section 5.5 remain valid for DisGB as well. In addition, however, DisGB uses geo-context information to select RPs: RPs can be either selected close to the publishers or subscribers of an event; which strategy is better depends on the scenario-specific workload. In general, selecting RPs close to the subscribers is better for workloads that involve many subscriber location updates, many subscription updates, and large subscription geofences that intersect with many broker areas as subscription information does not need to be distributed by the LB. Selecting RPs close to the publishers, on the other hand, is better for workloads that involve a high volume of published events as well as large event geofences that intersect with many broker areas, as the events can be matched at the LB of each publisher. We validate this assessment with experiments in Chapter 10.

6.7 Summary

In this chapter, we presented DisGB, a distributed pub/sub broker system that leverages geo-context for efficient inter-broker routing. DisGB builds upon GeoBroker (Chapter 5) and also uses two novel strategies for inter-broker message routing in multi-machine setups. As such, similarly to BCGroups (Chapter 4), we designed DisGB for distributed environments. While DisGB is more efficient than using broadcast groups (we prove this claim in Section 10.2), it can only play out its strength when very detailed IoT-specific domain knowledge, i.e., geo-context information, is available.

Chapter 7

MockFog: Automated Execution of Fog Application Experiments in the Cloud

In this chapter, we present MockFog, an approach for the automated execution of fog application experiments in the cloud. With MockFog, we can emulate an infrastructure testbed in the cloud to run fog application experiments. The testbed can be manipulated based on a predefined orchestration schedule. This allows application developers to evaluate the impact of sudden machine failures or unreliable network connections as part of a system test with varying load. Moreover, developers can test arbitrary failure scenarios and various infrastructure options at large scale.

We start in Section 7.1 with a high-level overview of MockFog's three modules. Then, in Section 7.2, we discuss how to use Mockfog in a typical application engineering process before describing each of the modules in Sections 7.3 to 7.5. In Section 7.6, we introduce our proof-of-concept implementation. Finally, we discuss and summarize this contribution in Section 7.7 and Section 7.8.

This chapter is based on material previously published in the Proceedings of ICFC 2019 [81] and our MockFog 2.0 paper that is still under review; a preprint has been published on arXiv [79].

7.1 MockFog Overview

MockFog comprises three modules: the infrastructure emulation module, the application management module, and the experiment orchestration module (see Figure 7.1).

For the first module, developers model the properties of their desired (emulated) fog infrastructure, namely the number and kind of machines but also the properties of their interconnections. The infrastructure emulation module uses this configuration for the infrastructure bootstrapping and infrastructure teardown. For the second module, developers define appli-

Figure 7.1: MockFog comprises three modules.

cation containers and where to deploy them. The application management module uses this configuration for the application container deployment, the collection of results, and the application shutdown. For the third module, developers define an experiment orchestration schedule that includes infrastructure changes and application instructions. The experiment orchestration module uses this configuration to initiate infrastructure changes or to signal load generators[22] and the system under test at runtime.

The implementation of all three modules is spread over two main components: the *node manager* and the *node agents*. There is only a single node manager instance in each MockFog setup. It serves as the point of entry for application developers and is, in general, their only way of interacting with MockFog. In contrast, one node agent instance runs on each of the cloud virtual machines (VMs) used to emulate the fog infrastructure. Based on the node manager's input, node agents manipulate their respective VM to show the desired machine and network characteristics to the application.

Figure 7.2 shows an example with three VMs: two are emulated edge machines, and one is a single "emulated" cloud machine.

In the example, the node manager instructs the node agents to manipulate the network properties of their VMs in such a way that an application appears to have all its network traffic routed through the cloud VM. Moreover, the node agents ensure that network manipulations do not affect communication to the node manager by using a dedicated management network. Note that developers can freely choose where to run the node manager, e.g., it could run on a developer's laptop or on another cloud VM.

[22] Requirements for load generators are highly application-specific and also depend on usage purposes such as benchmarking or testing. Since there is already a plethora of standard load generators, benchmarks, and application-specific ad-hoc load generators, we do not include a load generator in MockFog. Instead, we focus on integrating and managing arbitrary load generators through MockFog's signaling and orchestration features.

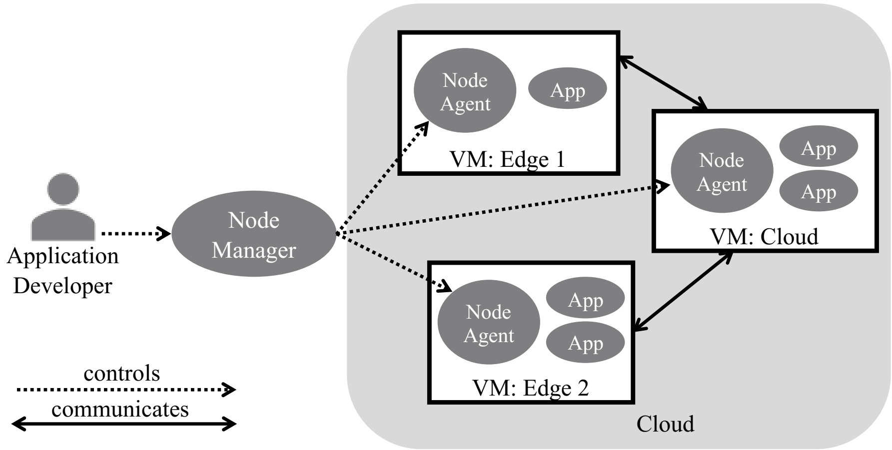


Figure 7.2: Example: MockFog node manager, node agents, and containerized application components (App).

7.2 Using MockFog in Application Engineering

A typical application engineering process starts with requirements elicitation, followed by design, implementation, testing, and finally maintenance. In agile, continuous integration and DevOps processes, these steps are executed in short development cycles, often even in parallel – with MockFog, we primarily target the testing phase. Within the testing phase, a variety of tests could be run, e.g., unit tests, integration tests, system tests, or acceptance tests [186] but also benchmarks to better understand system quality levels of an application, e.g., performance, fault-tolerance, or data consistency [18]. Out of these tests, unit tests tend to evaluate small isolated features only, and acceptance tests are usually run on the production infrastructure; often, involving a gradual roll-out process with canary testing, A/B testing, and similar approaches, e.g., [157]. For integration and system tests as well as benchmarking, however, a dedicated test infrastructure is required. With MockFog, we provide such an infrastructure for experiments.

We imagine that developers integrate MockFog into their deployment pipeline (see Figure 7.3) and use it with their existing continuous integration and deployment tooling. Once a new version of the application has passed all unit tests, MockFog can be used to set up and manage experiments. For the MockFog setup, a developer only needs to provide configuration files for the three MockFog modules, which we describe in more detail below. We provide the configuration files used within our evaluation in Chapter 11.

7.3 Infrastructure Emulation Module

A typical fog infrastructure comprises several fog machines, i.e., edge machines, cloud machines, and possibly also machines within the network between edge and cloud [17]. If no physical infrastructure exists yet, developers can follow guidelines, best practices or reference architectures

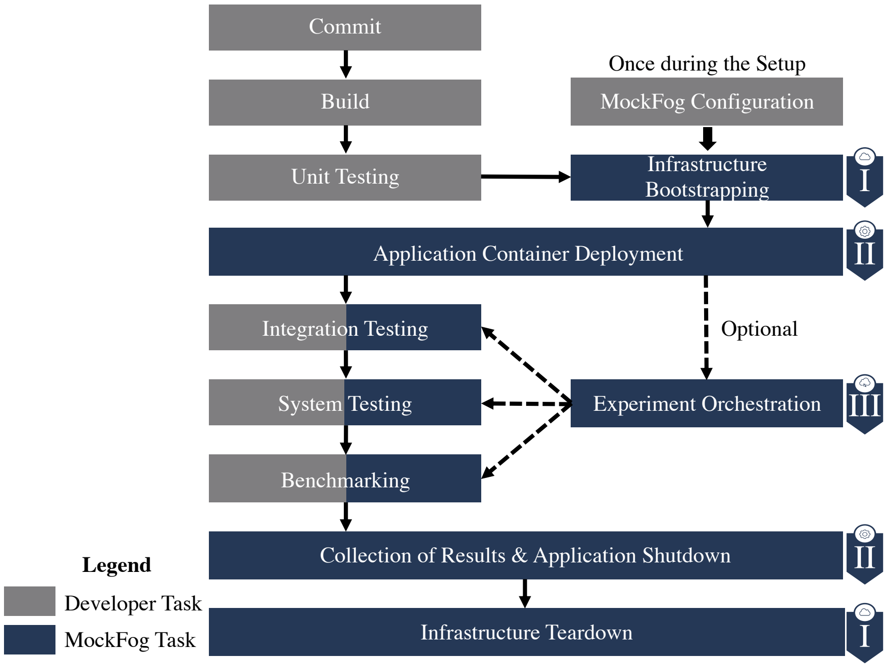


Figure 7.3: The three MockFog modules set up and manage experiments during the application engineering process.

such as proposed in [98, 131, 141, 151, 152]. On an abstract level, the infrastructure can be described as a graph comprising machines as vertices and the network between machines as edges [97]. In this graph, machines and network connections can also have properties such as the compute power of a machine or the available bandwidth of a connection. For the infrastructure emulation module, the developer specifies such an abstract graph before assigning properties to vertices and edges. We describe the machine and network properties supported by MockFog in Section 7.3.1 and Section 7.3.2.

During the infrastructure bootstrapping step (see Figure 7.3), the node manager connects to the respective cloud service provider to set up a single VM in the cloud for each fog machine in the infrastructure model. VM type selection is straightforward when the cloud service provider accepts the machine properties as input directly, e.g., on Google Compute Engine [61]. If not, e.g., on AWS EC2 [160], the mapping selects the smallest VM that still fulfills the individual machine requirements. MockFog then hides surplus resources by limiting resources for the containers directly. When all machines have been set up, the node manager installs the node agent on each VM, which will later manipulate its VM's machine and network characteristics.

Once the infrastructure bootstrapping has been completed, the developer continues with the application management module. Furthermore, MockFog provides IP addresses and access credentials for the emulated fog machines. With these, the developer can establish direct SSH connections, use customized deployment tooling, or manage machines with the cloud service provider's APIs if needed.

Once all experiments have been completed, the developer can also use the infrastructure emulation module to destroy the provisioned experiment infrastructure. Here, the node manager removes all emulated resources and deletes the access credentials created for the experiments.

7.3.1 Machine Properties

Machines are the parts of the infrastructure on which application code is executed. Fog machines can appear in various flavors, ranging from small edge devices such as Raspberry Pis[23], over machines within a server rack, e.g., as part of a Cloudlet [117, 155], to virtual machines provisioned through a public cloud service such as AWS EC2.

To emulate this variety of machines in the cloud, their properties need to be described precisely. Typical properties of machines are compute power, memory, and storage. Network I/O would be another standard property; however, we chose to model this only as part of the network in between machines.

While the memory and storage properties are self-explanatory, we would like to emphasize that there are different approaches to measuring compute power. AWS EC2, for instance, uses the amount of vCPUs to indicate the compute power of a given machine. This, or the number of cores, is a very rough approximation that, however, suffices for many use cases as typical fog application deployments rarely achieve 100% CPU load. It is also possible to use more generic performance indicators such as instructions per second (IPS) or floating-point operations per second (FLOPS). Our current proof-of-concept prototype (Section 7.6) uses Docker's resource limits [44].

7.3.2 Network Properties

Within the infrastructure graph, machines are connected through network connections: only connected machines can communicate. In real deployments, these connections usually have diverse network characteristics [182], e.g., slow and unreliable connections at the edge and fast and reliable connections near the cloud, which strongly affect applications running on top of them. Therefore, these characteristics also need to be modeled – see Table 7.1 for an overview of our model properties. For example, if a connection between machines A and B has a delay of 10 ms, a dispersion of 2 ms, and a package loss probability of 5%, a package sent from A to B would have a mean latency of 10 ms with a standard deviation of 2 ms and a 5% probability of not arriving at all.

In most scenarios, not all machines are connected directly to each other. Instead, machines are connected via switches, routers, or other machines. See Figure 7.4 for an example with routers and imagine having to model the cartesian product of machines instead.

[23] https://raspberrypi.org

Table 7.1: Properties of emulated network connections.

Property	Description
Rate	Available Bandwidth Rate
Delay	Latency of Outgoing Packages
Dispersion	Delay Dispersion (+/-)
Loss	Percentage of Packages Lost in Transition
Corruption	Percentage of Corrupted Packages
Reorder	Probability of Package Reordering
Duplicate	Probability of Package Duplication

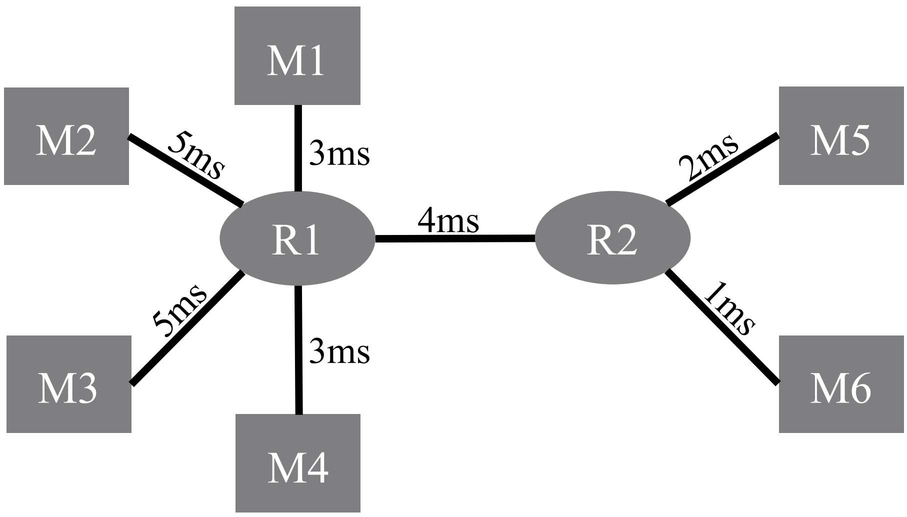


Figure 7.4: Example: Infrastructure graph with machines (M), routers (R), and network latency per connection.

In the graph, network latency is calculated as the weighted shortest path between two machines. For instance, if the connection between M2 and R1 (in short: M2-R1) has a delay of 5 ms, R1-R2 has 4 ms, and R2-M6 has 1 ms, the overall latency for M2-M6 is 10 ms. The available bandwidth rate is the minimum rate of any connection on the shortest path between two machines. The dispersion is the sum of dispersion values on the shortest path between two machines. Probability-based metrics, e.g., loss, are aggregated along the shortest path between two machines using basic probability theory methods ($p = 1 - \prod_{i=1}^{n}(1 - p_i)$).

7.4 Application Management Module

Fog applications comprise many components with complex interdependencies. The configuration of such an application also depends on the infrastructure as components are not deployed

```
{
    "container_name": "camera",
    "docker_image": "dockerhub/camera",
    "vm_directory": "/camera",
    "local_directory": "appdata/camera",
    "env": {
        "SERVER_IP": "{{ ip('cell-tower-2') }}",
        "SERVER_PORT": 8008
    },
    "command": [
        "--localRecording",
        "/camera/recording.mp4"
    ]
}
```

Listing 7.1: The container configuration comprises a unique container name and additional meta data.

on a single machine. For example, we need an IP address and port to communicate with a component running on another machine. MockFog can deploy and configure application components on the emulated infrastructure, resolving such dependencies. For this purpose, a requirement is that all application components are Dockerized. Furthermore, developers have to define application containers and how they should be deployed on the infrastructure. If these requirements cannot be met, developers can use their own deployment tooling instead of the application management module before continuing to the experiment orchestration.

For each container, the container configuration specifies a unique container name, the Docker image to be used, information on local files that should be copied to the VM from the machine of the node manager, environment variables, and command-line arguments. As an example, Listing 7.1 shows a very simple container configuration for a container with name *camera* in JSON format.

For this container, the Docker image is *dockerhub/camera*; if it is not available locally, MockFog pulls the latest version from Docker Hub. Furthermore, MockFog copies the contents of the local directory *appdata/camera* to the */camera* directory on each VM where the specific container will run. When the container is started, the environment variables *SERVER_IP* and *SERVER_PORT* are set to the specified values and become available to the application running inside the container. The value of *SERVER_IP* is resolved by a function that retrieves the IP address of the VM named *cell-tower-2*. There are more such functions, e.g., for retrieving the IP addresses of all VMs on which a container with a specific container name has been deployed. Finally, the *camera* container is instructed to write a local copy of its recording to */camera/recording.mp4* via command-line arguments. As this file path is inside the specified VM directory, its contents can be retrieved by MockFog automatically.

In the deployment configuration, developers specify for each container a deployment mapping of application components to VMs. Furthermore, they can also limit CPU and memory resources available to a container, e.g., for balancing the resource needs of multiple containers running on the same VM. During the application container deployment step (see Figure 7.3), the node manager installs dependencies on the VMs, copies files, and starts the configured containers.

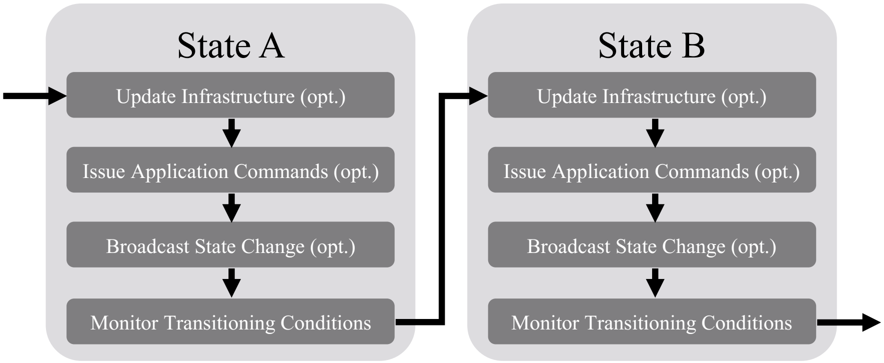


Figure 7.5: In each state, MockFog executes up to four actions; some actions are optional (opt.).

Once the experiment has been completed, the developer can also use the application container module for terminating the application and for collecting results.

7.5 Experiment Orchestration Module

There are various ways of testing and benchmarking an application. As discussed in Section 7.2, MockFog primarily targets integration and system tests as well as benchmarking because these require a dedicated test infrastructure. MockFog can artificially inject (and revert) failures to emulate network partitioning, simulate machine crashes and restarts, as well as other events for such experiments. This is particularly useful as failures are common in real deployments but will not necessarily happen while an application is being tested. Hence, artificial failures are the go-to approach for studying the fault-tolerance and resilience of an application [19]. While MockFog monitors the emulated infrastructure to detect deviations from what it configured, one might be interested in additional monitoring data. For this purpose, we recommend to either use the tooling of the chosen cloud vendor, e.g., Amazon CloudWatch [159] when running on AWS, or to deploy custom tooling, e.g., Prometheus [179], alongside the application through the application management module.

For the experiment orchestration step (see Figure 7.3), developers define an orchestration schedule in the form of a state machine. We describe the actions executed within a state in Section 7.5.1; we describe how developers can build complex orchestration schedules with states and their transitions in Section 7.5.2.

7.5.1 State Actions

The orchestration schedule comprises a set of states and a set of transitioning conditions. At each point in time, there is exactly one active state for which MockFog executes up to four actions in the following order (Figure 7.5):

Update Infrastructure (opt.) With MockFog, all properties of emulated fog machines and network connections (Table 7.1) can be manipulated. For this, the node manager parses the orchestration schedule and sends instructions to the node agents, which then update machine and network properties accordingly. For example, it is possible to reduce the amount of available memory (e.g., to mimic noisy neighbors), render a set of network links temporarily unavailable, increase network latency or package loss, or render a machine completely unreachable, in which case the node agent blocks all (application) communication to and from the respective VM. MockFog can also reset all infrastructure manipulations back to what was initially defined by the developer. Node agents acknowledge infrastructure updates to assert adherence to the orchestration schedule. If there are any problems that cannot be recovered autonomously, the node manager notifies the developer.

Issue Application Commands (opt.) Based on the orchestration schedule, the node manager can send customizable instructions to application components. For example, this can be used to instruct a workload generator to change its workload profile.

Broadcast State Change (opt.) It is sometimes necessary to notify application components or a benchmarking system of a new state being reached. While the Issue Application Commands action may distribute complex scripts if necessary, this action is a lightweight notification mechanism. In this, the first two actions are preparatory while this action signals to all components that the next experiment phase has been reached. This could, for instance, be logged to identify transitioning phases in experiment logs.

Monitor Transitioning Conditions Once the node manager reaches this action, an experiment timer is started. The node manager then continuously monitors if a set of transitioning conditions – as defined in the orchestration schedule – have been met. In MockFog, transitioning conditions can either be time-based or event-based: A time-based condition is fulfilled when the experiment timer reaches the specified time threshold. This is useful if a developer wishes to let the application run for a specific time to study effects of the active state. An event-based condition is fulfilled when the node manager has received the required amount of a specific event (messages). This is useful if a developer wishes to react to events distributed by application components, e.g., when any application component sends a *failure* event, MockFog should transition to an *ABORT EXPERIMENTS* state. If there are event-based conditions, application components have to either send events to the node manager directly or there must be a monitoring system such as Prometheus [179] from which the node manager can receive events.

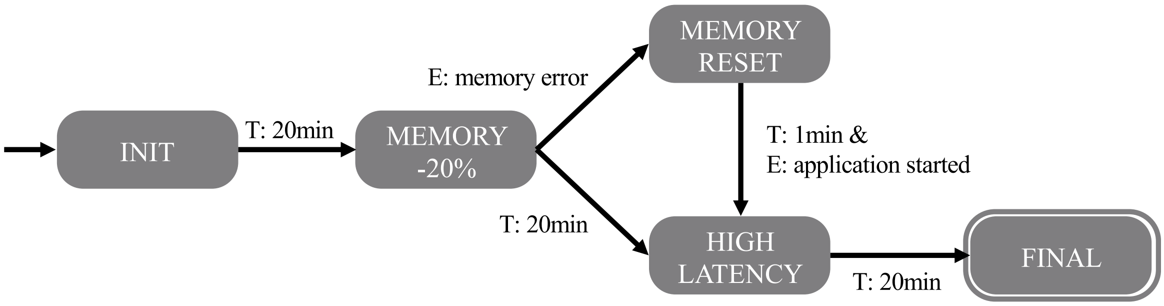


Figure 7.6: The experiment orchestration schedule can be visualized as a state diagram.

7.5.2 Building Complex Orchestration Schedules

For each state, developers can define multiple transitioning conditions; this allows MockFog to proceed to different states depending on what is happening during the experiment. For instance, an orchestration schedule could have a time-based condition that leads to an *ABORT EXPERIMENTS* state and additional event-based conditions that lead to a *NEXT LOAD PHASE* state. A transitioning condition may comprise several sub-conditions connected by boolean operators. This allows developers to define arbitrarily complex state diagrams, see for example Figure 7.6.

In the example, the orchestration schedule comprises five states; the arrows between states resemble the transitioning conditions. When started, the node manager transitions to *INIT*, i.e., it distributes the infrastructure configuration update and application commands. Afterward, it broadcasts state change messages (e.g., this might initiate the workload generation needed for benchmarking) and begins monitoring the transition conditions of *INIT*. As the only transitioning condition is a time-based condition set to 20 minutes (T: 20min), the node manager transitions to *MEMORY -20%* once it has been in the *INIT* state for 20 minutes. During *MEMORY -20%*, the node manager instructs all node agents to reduce the amount of memory available to application components by 20% via the Update Infrastructure action. Then it again broadcasts state change messages (e.g., this might restart workload generation) and starts to monitor the transitioning conditions of *MEMORY -20%*. For this state, there are two transitioning conditions. If any application component emits a *memory error* event, the node manager immediately transitions to *MEMORY RESET* and instructs the node agents to reset memory limits. Otherwise, the node manager transitions to *HIGH LATENCY* after 20 minutes. It also transitions to *HIGH LATENCY* from *MEMORY RESET* when it receives the event *application started* and at least one minute has elapsed. At the start of *HIGH LATENCY*, the node manager instructs all node agents to increase the latency between emulated machines. Then, it again broadcasts state change messages and waits for 20 minutes before finally transitioning to *FINAL*.

7.6 Proof-of-Concept Implementation

During the development of the MockFog approach, there were multiple MockFog prototype implementations. In this section, we describe our latest proof-of-concept implementation MockFog 2.0. MockFog 2.0 has been developed to achieve independence from specific IaaS cloud providers and can therefore be extended to the provider of choice. Our current open source proof-of-concept prototype[24] integrates with AWS EC2. By using EC2, MockFog has the same benefits and disadvantages as this cloud service: acquiring machines is easy and inexpensive, but experiments might be affected by factors such as busy neighbors. Thus, if higher emulation accuracy is needed, MockFog can easily be extended to also support other cloud services such as Grid'5000[25] or bare-metal machines. For this, one needs to add another Ansible playbook to the infrastructure module. Our implementation contains two NodeJS packages: the node manager (Section 7.6.1) and the node agent (Section 7.6.2).

7.6.1 Node Manager

The node manager NodeJS package can either be integrated with custom tooling or be controlled via the command-line. We provide a command-line tool as part of the package that allows users to control the three modules' functionality. For the infrastructure emulation module, the node manager relies on the Infrastructure as Code (IaC) paradigm. Following this paradigm, an infrastructure definition tool serves to "define, implement, and update IT infrastructure architecture" [122]. The main advantage of this is that users can define infrastructure in a declarative way with the IaC tooling handling resource provisioning and deployment idempotently. In our implementation, the node manager relies on Ansible [144] playbooks.

The node manager command-line tool offers several commands for each module. As part of the infrastructure emulation module, the developer can:

- Bootstrap machines: set up virtual machines on AWS EC2 and configure a virtual private cloud and the necessary subnets.
- Install node agents: (re-)install the node agent on each VM.
- Modify network characteristics: instruct node agents to modify network characteristics.
- Destroy and clean up: remove all resources and delete everything created through the *bootstrap machines* command.

When modifying the network characteristics for a MockFog-deployed application, the node manager accounts for the latency between provisioned VMs. For example, when communication

[24] https://github.com/MoeweX/MockFog2
[25] https://www.grid5000.fr/

should, on average, incur a 10 ms latency, and the existing average latency between two VMs is already 0.7 ms, the node manager instructs the respective node agents to delay messages by 9.3 ms. As part of the application management module, the developer can:

- Prepare files: upload the local application directories to the VMs and pull Docker images.
- Start containers: start Docker containers on each VM and apply container resource limits.
- Stop containers: stop Docker containers on each VM.
- Collect results: download the application directories from the VMs to a local directory on the node manager machine.

With the experiment orchestration module, the developer can initialize experiment orchestration. When the orchestration schedule includes infrastructure changes, the node manager instructs affected node agents to override their current configuration following the updated model. This is done via a dedicated "management network", which always has unmodified network characteristics and is hidden from application components.

7.6.2 Node Agent

While the node agent is also implemented in NodeJS, it uses the Python library `tcconfig` [181] to manage network connections. `tcconfig` is a command wrapper for the linux traffic control utility `tc` [20]. Thus, our current node agent prototype only works for Linux-based VMs. The node manager ensures that all dependencies are installed alongside the node agent.

The node agent can either be started by the node manager or manually via command-line. The only configuration necessary is the port on which the node agent exposes its REST endpoint. This REST endpoint is used by the node manager but can also be used by developers directly. To simplify its usage, we created a fully documented OpenAPI interface with Swagger [174].

Using the REST endpoint, one can retrieve status information and real-time `ping` measurements to a list of other machines. The node manager uses the `ping` measurement results to calculate the artificial delay, which should be injected to reach the desired latency between VMs. Furthermore, the endpoint can be used to set resource limits for individual containers as needed by the application management module for the *start containers* command and by the experiment orchestration module. Finally, the endpoint can be used to supply (and read the current) network manipulation configuration. On each update call, the node agent receives an adjacency list containing all other VMs. The list includes the corresponding specification of its effective metrics: how it should be realized from the viewpoint of the node manager's infrastructure model. If a particular machine should not be reachable, the adjacency list contains a package loss probability of 100% for the corresponding VM. This allows us to emulate network partitions easily.

7.7 Discussion

For the management of application containers, we decided to directly operate with Docker containers instead of using a more powerful solutions such as Kubernetes [4]. The main reason for this is that we do not want to assume that an application is using a certain container management solution; especially, because in practice different solutions are used in the cloud or at the edge. Since our MockFog prototype relies heavily on Ansible playbooks, one could easily add support for such solutions if necessary.

When an application relies on a managed Kubernetes service such as Amazon EKS[26], one should only use MockFog for emulating other parts of the infrastructure and set up a proxy machine that forwards traffic to EKS. This way, the cloud part of the application can run in its regular environment and MockFog only manages the non-cloud parts by changing network characteristics between the proxy and emulated edge machines.

When testing a large fog application, only the node manager is affected by increasing the number of VMs as it has to distribute instructions to more machines. In practice, however, the node manager will even for large-scale deployments be lightly loaded as distributing instructions is not particularly resource-intensive. Even in a situation where the node manager experiences a high load, it would only mean that state changes take slightly longer – the application itself would not be affected at all. It is also be possible to distribute the node manager. In practice, though, MockFog's scalability will usually be limited by the number of VMs that can be provisioned from the cloud provider of choice. Furthermore, it is – simply for cost reasons – not desirable to roll out a large-scale fog application to hundreds of MockFog nodes; it is also not necessary for testing and benchmarking purposes: When we visualize a fog environment as a tree with the cloud as the root node and edge devices as leaves, most paths from cloud to edge will run the same application components, e.g., in the smart factory use case, there might be multiple factories that send data to the cloud. For testing and benchmarking, however, it will usually suffice to only deploy a single example path on MockFog. Another option is to run groups of devices with similar network characteristics, such as multiple IoT sensors, on a few large VMs.

Finally, in its current state, MockFog's network manipulations only target the application layer. Thus, matters such as medium access contention, protocol specifications, e.g., enabling and disabling RTS/CTS for WIFI based networks, or specifics of mobile networks (4G/5G) are not emulated. To support such use cases, others have already come up with promising solutions in the context of MiniNet that might serve as a blueprint for extending MockFog: Fontes et al. [55] extended MiniNet to also emulate WIFI networks and Fiandrino et al. [3] added a mobile network suite to MiniNet. Also, even wired TCP/IP connections can be affected by other users, electrical interference, or natural disasters. For this, another solution could be to add a machine learning component to MockFog that updates connection properties based on past data collected on a reference physical infrastructure. Still, it is hard to justify this effort for most use cases.

[26] https://aws.amazon.com/eks/

7.8 Summary

In this chapter, we presented MockFog, an approach for the automated execution of fog application experiments in the cloud. With MockFog, application developers can emulate an infrastructure testbed in the cloud to run fog application experiments that can be manipulated based on a predefined orchestration schedule. In the emulated fog environment, virtual cloud machines are configured to closely mimic the real (or planned) fog infrastructure. This way, fog applications and fog systems can run in the cloud while experiencing comparable performance and failure characteristics as in a real fog deployment. With an emulated infrastructure, developers can also change machine and network characteristics, as well as the workload used during application testing at runtime based on an orchestration schedule. For example, this can be used to evaluate the impact of sudden machine failures or unreliable network connections as part of a system test with varying load. While testing in an emulated fog will never be as “good” as in a real production fog environment, it is certainly better than simulation-based evaluation only. Moreover, it allows application engineers to test arbitrary failure scenarios and various infrastructure options at large scale, which is also not possible on small local testbeds.

Part III

Experiments

In this part, we demonstrate the applicability of our approaches from Part II. For this, we carefully designed a set of experiments for each proposed system design. We conducted the experiments with the corresponding proof-of-concept system prototypes. Where appropriate, we complemented the experiments with simulation analyses.

In Chapter 8, we present the evaluation of BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems. In Chapter 9, we present the evaluation of GeoBroker, a single node pub/sub broker system leveraging geo-context for IoT data distribution. In Chapter 10, we present the evaluation of DisGB, an extension of the GeoBroker system that leverages geo-context to additionally improve inter-broker routing. Here, we also compare DisGB and BCGroups with simulation. Finally, in Chapter 11, we present the evaluation of MockFog, an approach for the automated execution of fog application experiments in the cloud. We also used MockFog concepts and (earlier versions of) the MockFog 2.0 prototype for the setup of execution environments in the Chapters 8 to 10.

Chapter 8

BCGroups

In this chapter, we present the evaluation of BCGroups. The first part of our evaluation is a simulation analysis (Section 8.1). It builds upon a global deployment to study the effects of the group formation process. The second part of our evaluation is based on experiments with our proof-of-concept prototype in an Internet of Vehicle (IoV) scenario (Section 8.2). These experiments only comprise a limited number of brokers as this is sufficient to validate the effectiveness of BCGroups: the strategy can manage the tradeoff between excess data and latency. We summarize our evaluation results in Section 8.3.

This chapter is based on material previously published in the Proceedings of ICFC 2020 [82].

8.1 Effects of the Group Formation Process

Many parameters influence the group formation process, e.g., the latency between broker machines, the latency threshold, heterogeneity of broker resources, or the number of broker instances. To better understand these effects, we implemented an event-discrete simulation[27] of the group formation process to evaluate it in large geo-distributed deployments. In the following, we will use simulation results to discuss:

1. how fast the group formation process terminates,
2. how the latency threshold influences the total number of broadcast groups,
3. and the group formation overhead.

We executed 160 simulation runs of the group formation process with different broker numbers and latency thresholds for this discussion. Broker locations and resulting network latency are based on the worldcities data set from simplemaps [175]. The intuition behind this is that each

[27] Our simulation tool is available on Github: `https://github.com/MoeweX/broadcast-group-simulation`

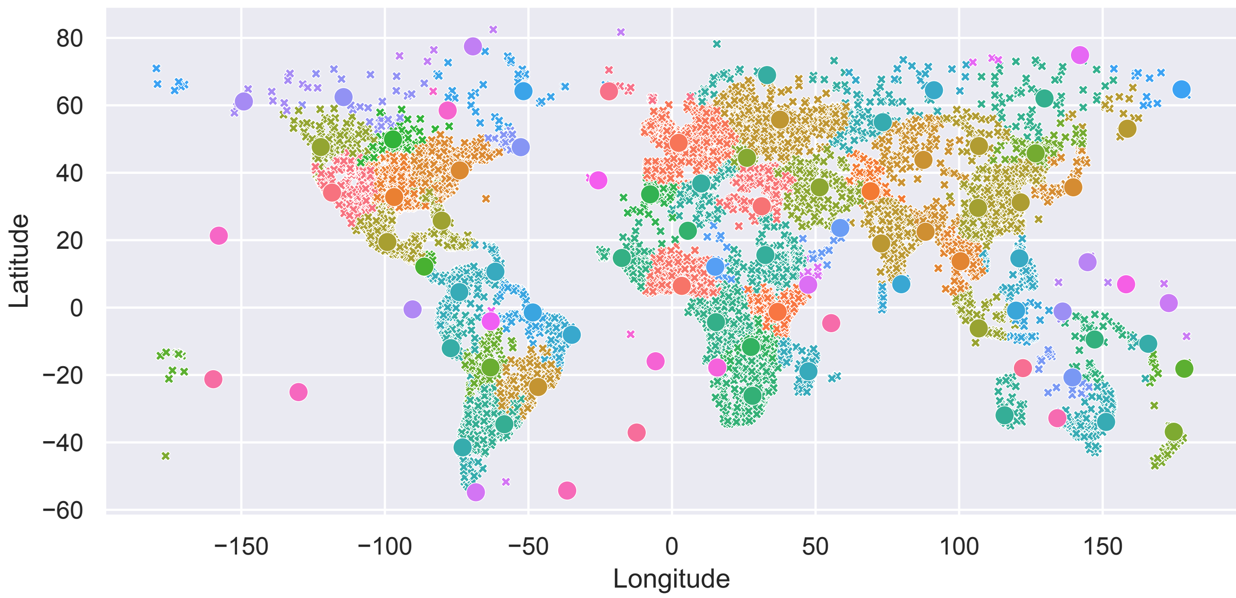


Figure 8.1: A simulation run with 12,000 brokers and a latency threshold of 30ms led to 89 broadcast groups. Leaders (circle) and members (cross) of the same group have the same color.

city has access to its own pub/sub broker and that all these fog brokers are inter-connected for global communication. The latency between two brokers is calculated by multiplying their physical distance by 0.021 ms/km. We determined this constant by analyzing the 2016 IPlane traceroute dataset [116]. Figure 8.1 visualizes the results of one simulation run, where the setup led to 89 leaders, e.g., in Los Angeles, New York, or Tokyo.

Our simulation is based on ticks, during each tick:

- Leaders negotiate a group merge with the closest other leader (we use the simple LCM approach[28]).
- Leaders notify their members if they join the broadcast group of another leader.
- Notified members join the new leader if latency permits, or start their own broadcast group.

The next tick is only started when all brokers have completed all actions of the current tick. We can use the number of ticks as a notion of the time required to complete the group formation process, as it does not depend on the simulation's execution environment. From Figure 8.2, one can see that the time needed to complete the group formation process scales linearly with the number of brokers. Furthermore, decreasing the latency threshold also slightly increases the group formation time.

Figure 8.3 shows that increasing the latency threshold decreases the total number of broadcast

[28]Leadership Capability Measure, see Section 4.2.

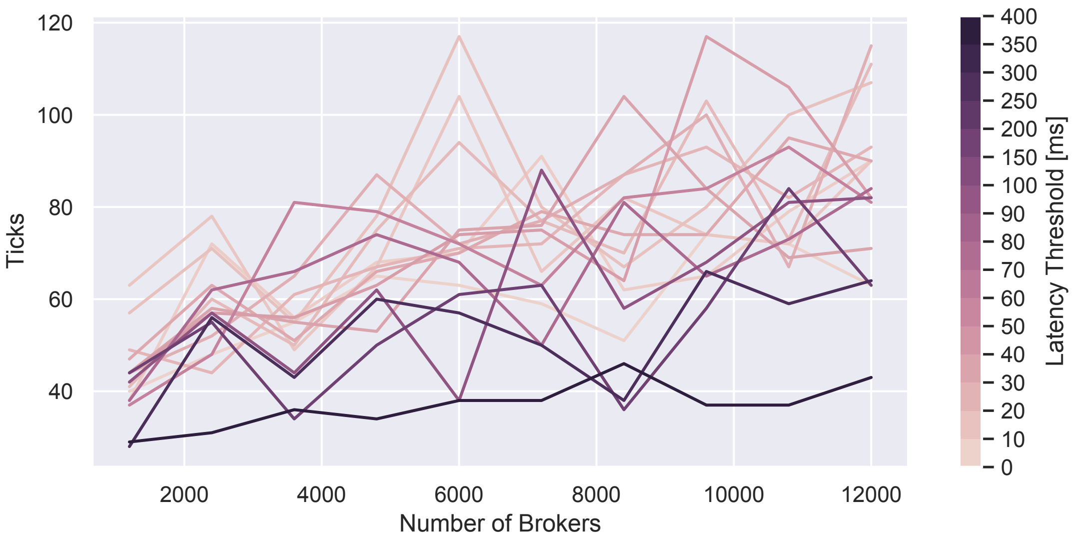


Figure 8.2: The time needed to complete the group formation process scales linearly with the number of brokers.

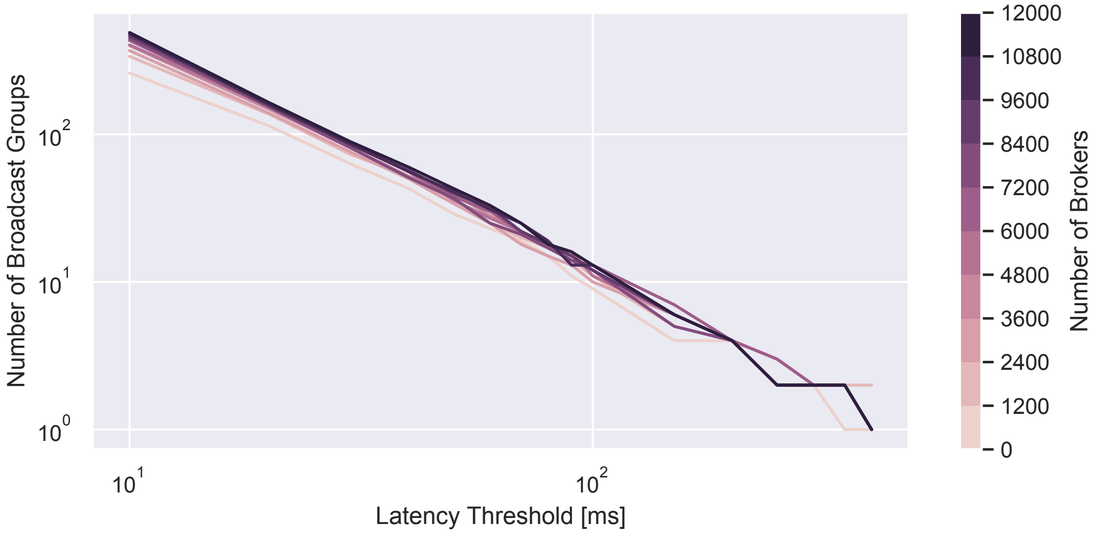


Figure 8.3: Latency thresholds control the amount of broadcast groups (logarithmic scale).

groups quadratically (note the logarithmic scales in the figure). In addition, the total number of brokers does not influence this result for higher broker numbers. This is expected and confirms the effectiveness of our threshold-based approach: we can control the number of groups and thus the average group size with the latency threshold.

Figure 8.4 shows that the number of leader and member join operations needed to reach a stable state scales linearly with the number of brokers—also for different latency thresholds. The minimum number of messages[29] required to negotiate a leader join is three (join request with LCM, join reply with proposal for new leader, actual leader join message). The minimum

[29] More messages might be required if messages are acknowledged or a different negotiation protocol is used.

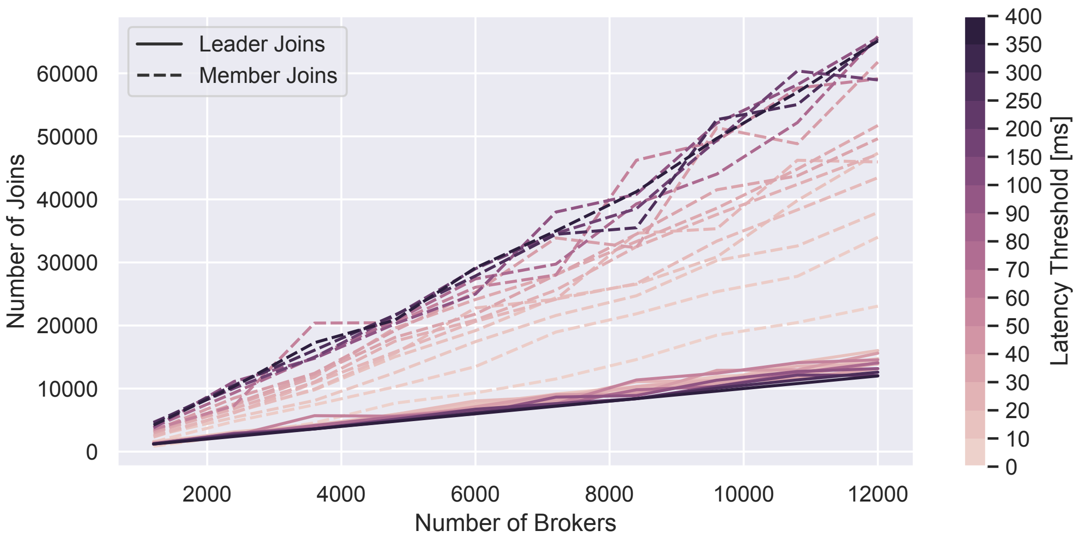


Figure 8.4: The number of operations scales linearly with the number of brokers.

number of messages for a member join is two (member notification message, actual member join message). In our simulation, on average, 1.04 member notification messages have been sent for every member join message, as members only join if the connection to the new leader has a latency below the threshold. As the number of messages increases linearly with the number of operations, the required message overhead of the group formation is $\mathcal{O}(N)$.

Infrastructure changes such as the addition or removal of a broker also trigger the group formation process. However, the process is then considerably shorter than the initial one and depends on the extent of change. In the best case, a new broker simply joins another leader while all other brokers are not affected.

Note, that for the group formation process, brokers rely on latency measurements to other brokers. The number of required measurements depends on the number of brokers and leaders: If N is the total number of brokers, the process involves $(\mathcal{O}(N-L+L^2))$ measurements; $N-L$ measurements from members to their leaders and L^2 measurements between leaders. If the latency threshold is set to 0, every broker would be a leader (similar to cloud relay), so $L = N$. This leads to $\mathcal{O}(N-N+N^2) = \mathcal{O}(N^2)$ measurements in the worst case. If the latency threshold is set to a very large value, all brokers end up in the same broadcast group (similar to event flooding), so $L = 1$. This leads to $\mathcal{O}(N-1+1^2) = \mathcal{O}(N)$ measurements in the best case. When using a latency threshold that results in a middleground solution, e.g, $L = \sqrt{N}$, the number of measurements is still $\mathcal{O}(N-\sqrt{N}+(\sqrt{N})^2) = \mathcal{O}(N)$. The number of needed latency measurements can be additionally reduced by partitioning leader announcements, as explained in Section 4.2.

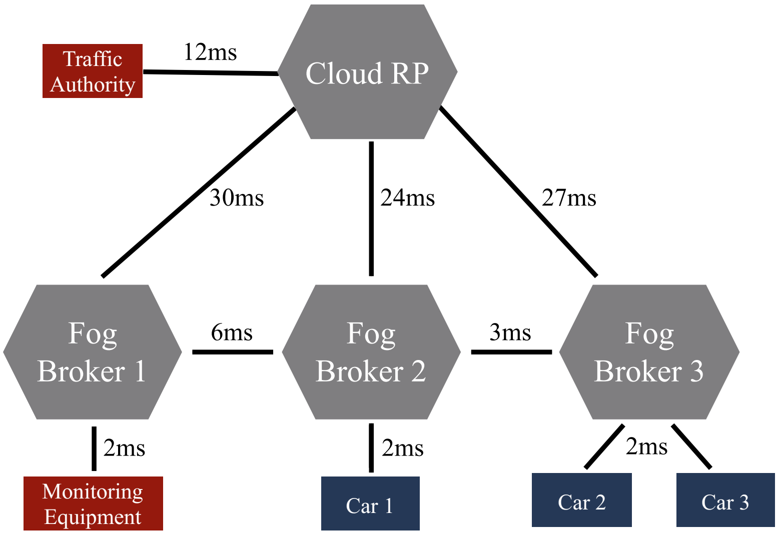


Figure 8.5: Evaluation setup.

8.2 Effectiveness of BCGroups

To evaluate the effectiveness of BCGroups, we ran experiments based on an IoV scenario. During the experiments, we collected data on latency and excess data dissemination. As execution environment, we used an emulated infrastructure with multiple broker instances, which we bootstrapped with MockFog (see Chapter 7).

Scenario: Our scenario is a simplified IoV use case with three types of clients: cars, monitoring equipment, and traffic authorities. For improved driving safety, cars exchange telemetry data with other cars so that they know when cars brake or change lanes. The monitoring equipment, e.g., a camera, collects traffic information that it sends to the traffic authority for processing. The traffic authority could use the collected data to inform cars about traffic jams or accidents; we refrained from doing so to keep the use case simple.

Evaluation Setup: Our evaluation setup can be seen in Figure 8.5. We deployed three of our proof-of-concept brokers in the fog, to which a total of four clients (three cars and one piece of monitoring equipment) connected. Because our brokers still support the MQTT protocol, we can use the standard Mosquitto command-line clients for communication. As cloud RP, we deployed a vanilla Mosquitto MQTT broker; every other system that supports the MQTT protocol could be used as well.

We ran experiments with three inter-broker routing strategies: BCGroups, event flooding (see Section 2.3.3), and central-RP (here, there is a single RP (see Section 2.3.3) that matches all events and subscriptions that cannot be matched at the local broker). We chose these strategies because event flooding minimizes latency but leads to a lot of excess data, and central-RP minimizes excess data at the cost of latency. BCGroups, on the other hand, aims to balance latency and excess data dissemination.

Infrastructure and Deployment: As we do not have access to a fog computing infrastructure with the characteristics shown in Figure 8.5 available, we used MockFog to emulate such an infrastructure. Based on our infrastructure definition, MockFog deployed one virtual machine for each component on AWS EC2 [160], configures networking delays, and deploys the brokers and clients. We used `t3.small` instances for brokers and `t3.nano` instances for clients.

Experiment Execution: For each of the three communication strategies, event flooding, BCGroups, and central-RP, we ran the same 15-minute workload: For the exchange of telemetry data, each car publishes 20 bytes of data to a unique topic 20 times per second, e.g., Car 1 publishes to the topic */car-telemetry/realtime/1*. At the same time, all cars subscribe to the wildcard topic */car-telemetry/realtime/+* which matches the individual topics. The traffic authority collects data from monitoring equipment by creating a subscription to the topic */traffic-control/monitoring/#*. The monitoring equipment publishes its data (1000 bytes, once per second) to the (unique) topic */traffic-control/monitoring/1*.

The experiments were run with our proof-of-concept prototype (see Section 4.3). For the event flooding strategy, we set up MQTT bridging [7] between all brokers, which is supported by Moquette out of the box. As a consequence, each message received by one broker is forwarded to every other broker. For BCGroups, we used a latency threshold of 5 ms. This leads to two broadcast groups, one that comprises only Fog Broker 1, and one that comprises Fog Broker 2 and Fog Broker 3. For central-RP, we set the latency threshold to 0 which leads, in our setup, to three broadcast groups that each comprise a single fog broker.

In the following, we discuss the measured message delivery latency (Section 8.2.1) and amount of produced excess data (Section 8.2.2) for each strategy.

8.2.1 Message Delivery Latency

The message delivery latency (MDL) for each individual message is defined as $\text{MDL} = t_{\text{received}} - t_{\text{send}}$, with t denoting the send and receive timestamp measured at each client. Since AWS EC2 machines have highly synchronized clocks [164], clock drift is negligible and can, thus, be disregarded for our experiments.

Considering the topology, MDL for event flooding and BCGroups should be comparable. For both strategies, we expect 50% of the messages received by Car 2 and Car 3 to have an MDL of about 4 ms, as the corresponding messages can be matched at Fog Broker 3 directly and the one-way latency between Fog Broker 3 and each car is 2 ms. The remaining 50% of the messages received by Car 2 and Car 3, as well as all messages received by Car 1, should have an MDL of about 7 ms, as the messages additionally need to be sent via the link between Fog Broker 2 and Fog Broker 3. Figure 8.6 shows the experiment results which confirm our expectation. In particular, there is a clearly visible step when the cumulative distribution reaches 50% for Car 1 and Car 2, which confirms the general performance of BCGroups[30].

[30]The latency values are not exactly 4 ms (and 7 ms), as processing messages on each fog broker also requires time.

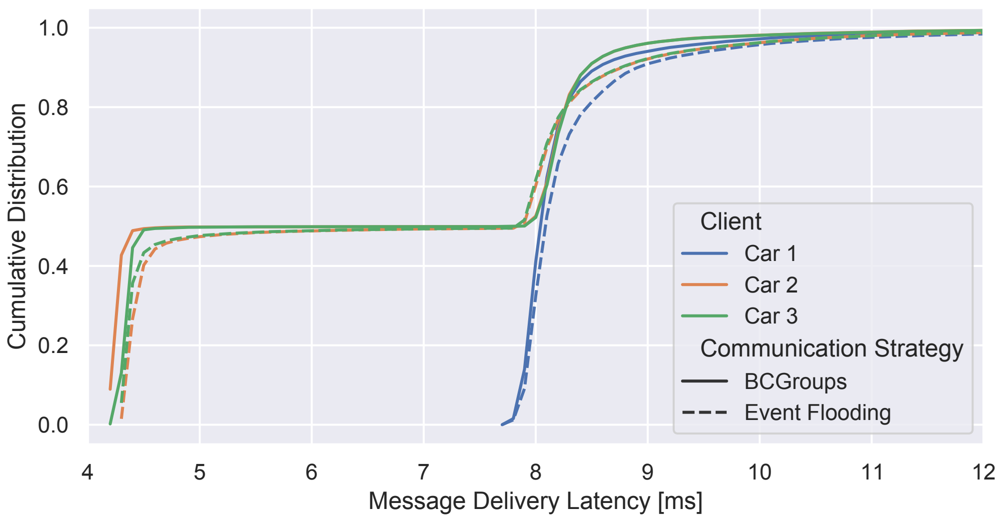


Figure 8.6: Car Telemetry MDL for event flooding and for BCGroups.

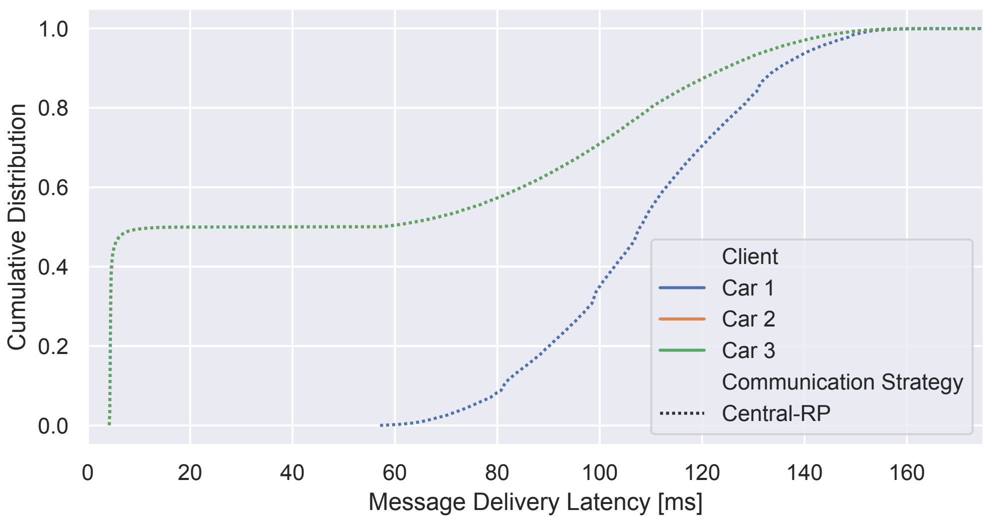


Figure 8.7: Car Telemetry MDL for central-RP (note, lines for Car 1 and Car 2 overlap).

For the central-RP strategy, we expect 50% of the messages received by Car 2 and Car 3 to have a similar MDL as in the event flooding or BCGroups experiment. The remaining 50% of their received messages, as well as all messages received by Car 1, should be routed via the cloud, which leads to an MDL of 55 ms or higher. Figure 8.7 shows that the used implementation achieves this latency for some messages. However, this experiment also reveals that our setup with the Eclipse Paho MQTT client library, which is used by the Moquette fog brokers to create subscriptions at the Mosquitto cloud broker, negatively influences MDL (visible by the long-tail)[31].

[31] This is not visible in any other experiment, as only here a subscription created from one broker (Fog Broker 2 and Fog Broker 3) at another broker (Cloud RP) matches incoming messages. Intra-group communication is done via broadcasting, which does not depend on subscriptions.

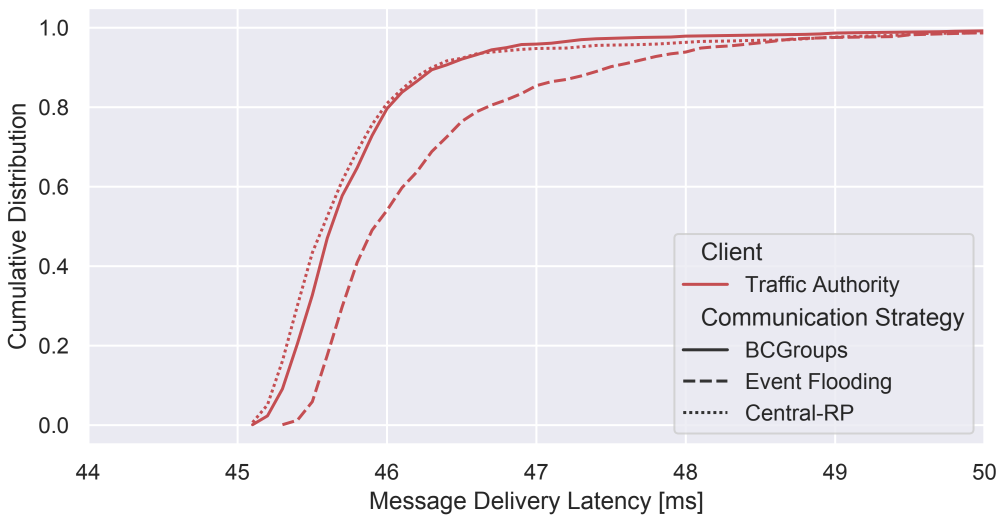


Figure 8.8: Monitoring data MDL.

All messages received by the traffic authority, i.e., all messages published by the monitoring equipment, should have a 44 ms latency. Figure 8.8 shows that, considering the mentioned overheads, the results match our expectation. Note that our forwarding implementation seems to have a better performance than Moquette's bridging which is used by the event flooding strategy.

In conclusion, BCGroups can achieve a communication latency close to the one of event flooding. Depending on the workload, this can be significantly better than the communication latency achieved by central-RP.

8.2.2 Excess Data

In the following, we evaluate the impact of each communication strategy on excess data dissemination. For that, we logged each message processed by every broker and determined the amount of correct and redundant messages. Correct messages are messages processed by brokers that either have a matching subscriber or to which the publishing client is connected. Redundant messages are all other messages processed by brokers; these messages have to be discarded and therefore count as excess data.

Tables 8.1 and 8.2 show the amount of correct and redundant messages, as well as the share of excess data, for each communication strategy and both message flows.

The car telemetry messages have to be processed by Fog Broker 2 and Fog Broker 3 only; thus, all car telemetry messages processed by Fog Broker 1 and Cloud RP are excess data. The monitoring messages have to be processed by Fog Broker 1 and Cloud RP only; thus, all monitoring messages processed by Fog Broker 2 and Fog Broker 3 are excess data. While we

Table 8.1: Excess data for each communication strategy: car telemetry

	Event Flooding	Central-RP	BCGroups
Correct Msgs.	107948	107944	107944
Redundant Msgs.	107948	53972	53972
Excess Data	50%	33%	33%

Table 8.2: Excess data for each communication strategy: monitoring data

	Event Flooding	Central-RP	BCGroups
Correct Msgs.	1798	1798	1798
Redundant Msgs.	1798	0	0
Excess Data	50%	0%	0%

used the same workload for all three communication strategies, the respective message numbers are not exactly identical due to minimal runtime variations of our publishing clients.

In conclusion, event flooding results in the largest amount of excess data, while BCGroups achieves a similar efficiency as central-RP.

8.3 Summary

In summary, we demonstrated that BCGroups achieves results in terms of communication latency and excess data that can be considered the best of both worlds, effectively balancing the tradeoff. In addition, running the group formation process even with high broker numbers is possible due to a message overhead of $\mathcal{O}(N)$. BCGroups can also be used to mimic the two other evaluated strategies: For event flooding, the latency threshold can be set to a very large value so that all brokers end up in the same broadcast group. To use a central RP, the latency threshold can be set to zero so that every broker creates its own broadcast group. This observation also emphasizes that BCGroups indeed provides a way to manage the tradeoff between excess data and latency.

Chapter 9

GeoBroker

In this chapter, we present the evaluation of GeoBroker. We used the Geolife dataset [194] to generate realistic workloads (Section 9.1) for the evaluation of the GeoCheck overhead (Section 9.2) and for a use case evaluation (Section 9.3). We summarize our evaluation results in Section 9.4.

This chapter is based on material previously published in the Proceedings of ISYCC 2019 [76], in the Computer Communications journal [75], and the Software Impacts journal [74].

9.1 Generating a Realistic Workload

To evaluate the performance and the overhead of GeoBroker, we need a dataset with spatial information. We chose the Geolife V1.3 data set [194] which contains 18,670 GPS trajectories collected over five years. More than 90% of the trajectories contain one entry every one to five seconds, and each entry comprises a timestamp and the user's current location.

Based on the data set, we implemented two types of clients for different workloads, *TravelClient* and *TeleportingClient*. The ClientManager (see below) is responsible for starting both types of clients. When a client is started, it is initialized with one trajectory from the data set and keeps "traveling" along the corresponding route of locations until it is shut down. When a client arrives at a location, it executes several pre-defined operations; what these operations are depends on the desired workload type, e.g., updating a subscription or publishing an event. The TravelClient uses the timestamps to determine how much time it takes to arrive at the next location; the TeleportingClient ignores the timestamps and processes the trajectory as fast as possible. When the last location of a trajectory has been reached, both types of client immediately jump back to the first location of their trajectory and start to travel again. Each client runs in its own thread, thus, implementing a closed workload model [18].

For the evaluation, three operations are of particular interest: First, updating the current client

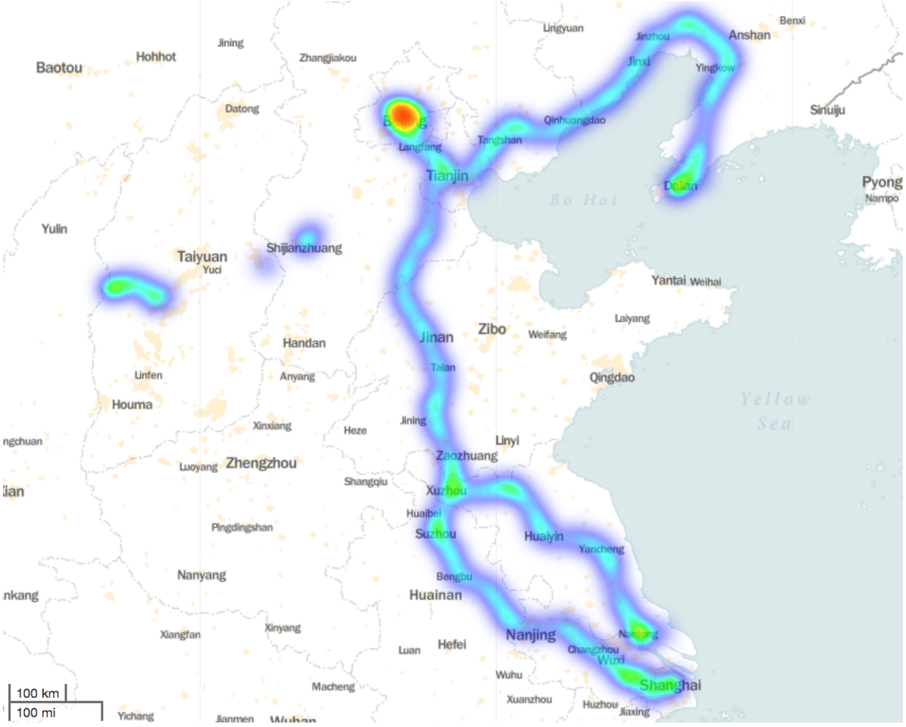


Figure 9.1: Location-heatmap for the first 1000 trajectories.

location. Second, updating subscriptions for a given client; this includes removing the old one first if a client has already created one for a specific topic. Third, publishing an event.

We also implemented a ClientManager which starts clients and assigns trajectories; multiple ClientManagers can be synchronized[32] by providing a common start time. To ensure determinism, we assigned an incrementing identifier to each trajectory in the dataset. For example, if clients for the first 1000 trajectories should be started, plotting the trajectories' locations in a heatmap always yields the picture shown in Figure 9.1.

9.2 GeoCheck Overhead

With the following experiments, we want to quantify the overhead on the subscription indexing structure when using GeoChecks compared to solely using ContentChecks. We do this by reporting the operation throughput for subscription **update** operations (a subscriber updates its subscription) and subscription **get** operations (a publisher publishes an event and GeoBroker needs to determine the subscribers) when GeoChecks are enabled (GEO) and disabled (NoGEO). These experiments were run with our subscription indexing structure only rather than using the complete GeoBroker implementation, as we do not want networking, message encoding/decoding, and other factors to influence our results. The overhead highly depends on the implementation and type of indexing structure. Therefore, this experiment primarily serves the

[32] Synchronizing ClientManagers is necessary when more than one machine is used as each ClientManager and its clients run on the same machine.

Table 9.1: Parameters of the GeoCheck overhead experiment.

Parameter	Set of Evaluated Values
Number of Clients	1, 10, 100, 250, 500, 750, 1000
Update/Get Ratio	(99/1), (1/1), (1/10), (1/99)
GeoCheck	GEO and NoGEO
Granularity	1, 10, 25, 50, 100
Total Number of Runs	56

purpose of getting a general understanding of parameters resulting in the highest GeoCheck overheads, proving that our implementation is efficient enough, and putting the results of the following use case evaluation into perspective.

Table 9.1 shows our evaluated parameter set. The update/get ratio describes the ratio of update and get operations, e.g., (1/10) means that each update operation is followed by ten get operations. All experiments have been run on a single t3.xlarge AWS EC2 [160] instance configured in unlimited mode for 15 minutes each. During each experiment, the clients continuously send update or get messages in compliance with the update/get ratio to the subscription data structure. Whenever they reach a location, they execute one operation; thus, we use the TeleportingClient to send requests as fast as possible. In the NoGEO runs, clients update a single subscription to an example topic and get all subscriptions with a matching topic. In the GEO runs, clients additionally supply a circular subscription geofence around their current location with each subscription (radius = 0.01 degree which is roughly 1km at this latitude/longitude) and get only subscriptions with a matching topic based on their current location.

As explained in Section 5.3, the granularity value can be used to tune the subscription indexing structure performance (for the GEO runs); depending on the average geofence size of subscriptions, as well as the update/get ratio, different values yield the best result. Figure 9.2 shows the number of operations that can be processed every second for 10 clients simultaneously. When the raster granularity increases, the raster fields become smaller which improves get performance. At the same time, this impairs update performance as each subscription must be added to more raster fields. Note that increasing the granularity value has a similar effect on performance as increasing the geofence size, as the computational effort of Algorithm 2 on page 47 depends on the number of raster fields inside the bounding box. Thus, we did not run additional experiments with different geofence sizes.

To compare GEO to NoGEO runs, we set the raster granularity to 25 as this value has a good performance for all four update/get ratios for the chosen geofence size. Figure 9.3 shows how using geo-context information affects the performance of the subscription data structure. Positive values mean that the GEO throughput is X times higher than the NoGEO throughput; negative values indicate the opposite. If there is only a single client, the throughput is 24.37 times higher in the NoGEO run with the (99/1) update/get ratio, and 1.13 times higher for the

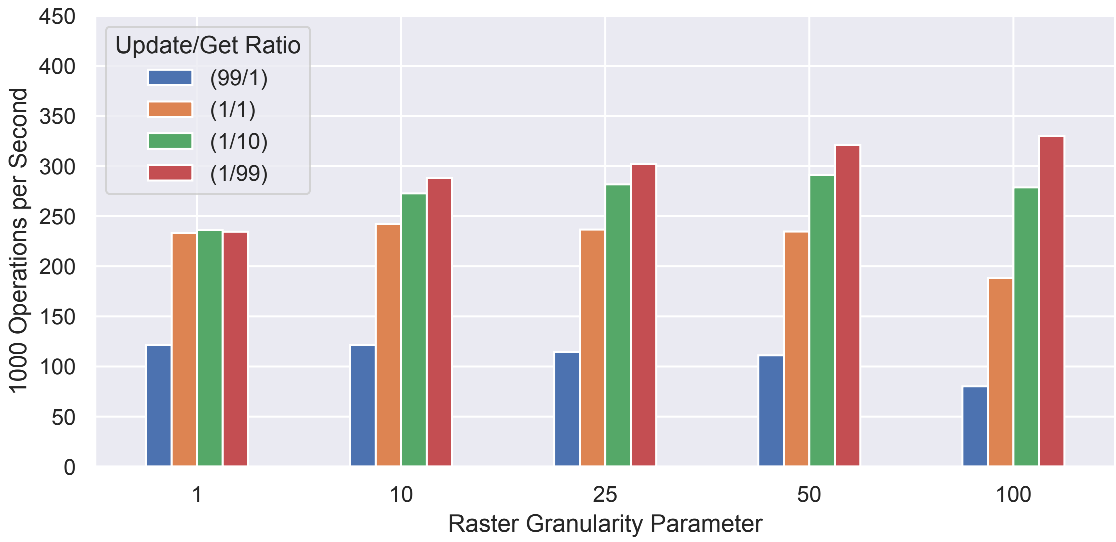


Figure 9.2: GEO operation throughputs for different granularity values.

Figure 9.3: Using geo-context information increases the performance of the subscription indexing structure for publish-heavy workloads.

(1/99) ratio, as updating the geofences in our storage component is more expensive than retrieving them. However, this picture changes for higher client numbers, as NoGEO gets always return all existing subscriptions with matching topics while GEO only returns the subscriptions of nearby clients with a matching topic leading to higher throughputs for publish-heavy workloads. Thus, using geo-context information can help to significantly decrease the load on clients (irrelevant events are simply not delivered) while also increasing the broker's performance.

Aside from the experiments above that evaluate the overhead on the subscription indexing structure for different configurations in a realistic scenario, we also determined the overhead of processing published events when every event is always received by all clients (GEO and

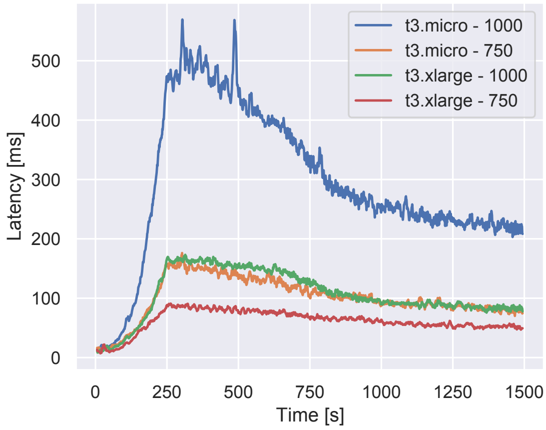
NoGEO). For this, the ContentCheck and both GeoChecks must always be true for each connected client and published message. This means that in GEO runs both geofences must be very large to comprise all clients. While using such large geofences is not realistic, as then checks could be omitted altogether, it allows us to determine an upper bound on the GeoCheck overhead. We created an artificial workload; here, 100 clients create a single subscription and then publish a total of 11,547 events to the same topic for 850 seconds. We ran this workload on a single t3.large instance that hosts the clients and the (full) broker. To compare the performance between GEO and NoGEO, we measured the message delivery latency (MDL) for each message: MDL $= t_{received} - t_{send}$, with t denoting the send and receive timestamp measured at each client. The average MDL in the GEO run is 4.08 ms, the average MDL in the NoGEO run is 3.79 ms. Thus, **the overhead of doing the two GeoChecks is about 7.7%** for this artificial workload. Note that the granularity does not matter for this experiment since clients only create a single subscription and both GeoChecks are run on every subscription.

9.3 Application Use Case

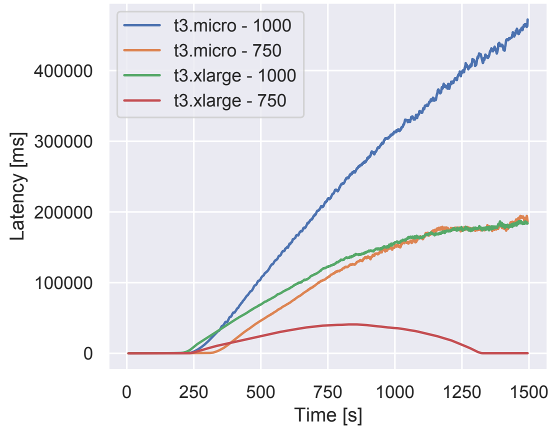
With this experiment, we want to show how GeoBroker behaves in a realistic use case and analyze its performance in GEO and NoGEO runs, i.e., when using geo-context information is enabled and disabled. In our use case, clients travel on their route and publish events to all other clients in close proximity when reaching a new location. The events could contain any data, e.g., surface condition information (roughness, surface, slipperiness), an image of the surroundings, broadcasted text messages, or requests for assistance.

In the GEO experiments, the subscription geofence ensures that only events from nearby publishers are received, while the event geofence ensures that data is not sent to subscribers outside a defined area, e.g., advertisement companies collecting user data. For this, TravelClients connect to GeoBroker and execute the following three operations each time a location has been reached: First, send a ping message to update the current location. Second, create/update the subscription to the topic “data” with a new circular subscription geofence (radius = 0.01 degree) around the current location. Third, publish a new event to the topic “data” with a payload size of 750 bytes and an event geofence (radius = 0.01 degree) that also surrounds the current location. Thus, each client acts as a publisher and as a subscriber at the same time. In the NoGEO experiments, a subscription is created only once, and thereafter events are published without a geofence. This means that clients do not have to update their subscriptions or locations, so we also skip ping messages for NoGEO.

We ran each GEO and NoGEO experiment for 25 minutes with 250, 500, 750, and 1000 clients (250 clients are running together on one t3.xlarge instance). We first ran all experiments with GeoBroker deployed on a t3.xlarge instance and then repeated them with GeoBroker deployed on a t3.micro instance to study how GeoBroker copes with limited computational resources. During the GEO experiments, update and get operations are alternating as we want the clients to update their subscription and publish an event each time they arrive at a location.



(a) GEO



(b) NoGEO

Figure 9.4: Publish latency for 750 and 1000 clients on t3.micro and t3.xlarge.

As for the corresponding (1/1) update/get ratio the subscription data structure has the highest throughput when the granularity is set to 10 (see Figure 9.2); we also set the granularity to 10 for this use case evaluation.

We measured the publish latency for each test run, which is the time between publishing an event and receiving an acknowledgment that the event has been sent to all subscribers that passed the ContentCheck and both GeoChecks (see Section 5.2). As the event might still be on the wire, all subscribers may not yet have received it. In general, however, the publish latency can be expected to be close enough to the delivery latency. Furthermore, as this latency can be measured on the sending client's machine, it is not affected by clock synchronization issues.

The first observation is that NoGEO only works for 250 and 500 clients (and for 750 clients on t3.xlarge). The latency continues to increase for more clients – up to several minutes – rather than stabilizing at a certain value (see Figure 9.4b). For GEO, on the other hand, the publish latency remains below a second, even for 1000 clients on a t3.micro instance (see Figure 9.4a).

Furthermore, the message loss is substantial for NoGEO runs with many clients, i.e., the t3.micro broker lost at least 22.5% of its messages in 25 minutes with 750 clients, and 45.1% of its messages with 1000 clients due to being overloaded. This observation makes sense when studying the number of delivered messages[33] at 250 and 500 clients. For 250 clients, the NoGEO broker delivers 9.4 million messages while the GEO broker delivers 3.6 million messages. This also means that for 250 clients more than 60% of the transmitted messages have no relevance to receiving clients. For 500 clients, the NoGEO broker delivers 39 million messages (GEO = 8.9 million messages). When inspecting the CPU load (see Figure 9.5), one can also identify that t3.micro NoGEO has a substantial amount of dropped messages including many connect and subscribe messages during the startup phase as the CPU load does not significantly increase for 1000 clients compared to 750 clients.

[33]Includes acknowledgements messages and event deliveries.

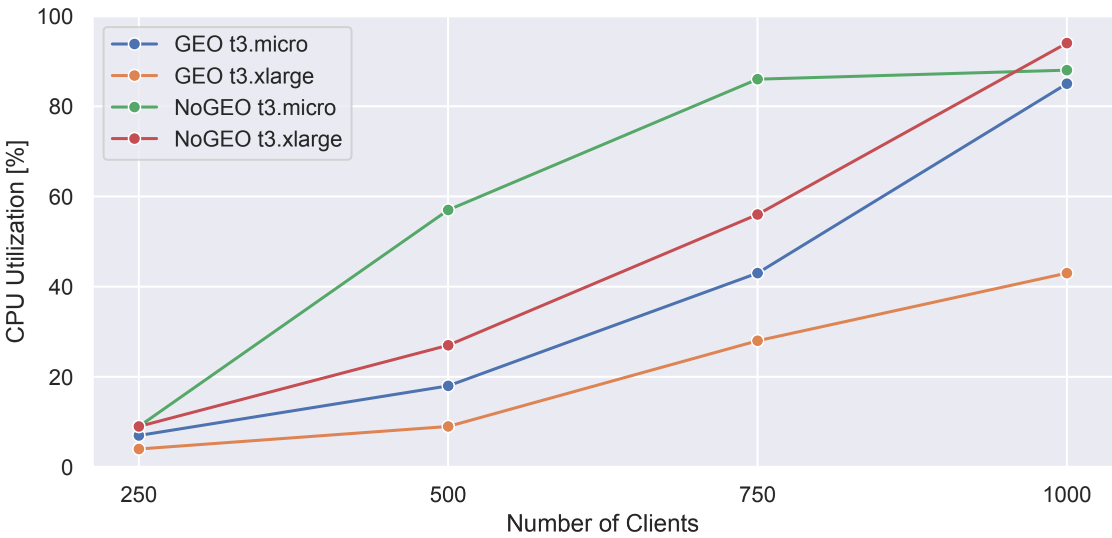


Figure 9.5: The CPU load is considerably higher for NoGEO than for GEO.

In general, **GEO latency is always lower than NoGEO latency, even though additional computations are necessary**. The only exception is the 250 client experiment on the t3.xlarge instance: here, the NoGEO publish latency is on average 51ms, compared to 57ms for the GEO experiment. Figures 9.6 and 9.7 show the average publish latency over time for the GEO and NoGEO run with 250 and 500 clients on t3.micro[34].

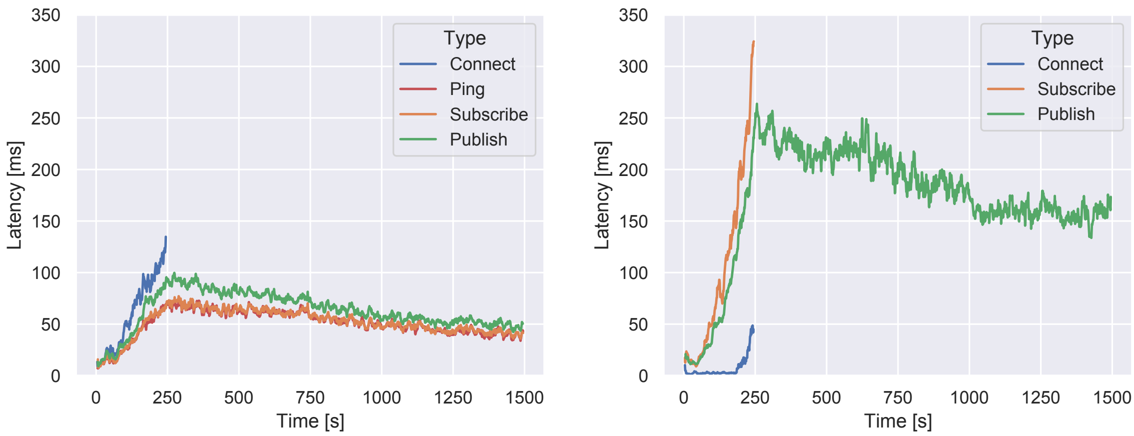
Besides the publish latency, the figures also contain the connect, ping, and subscribe latency (latency between sending a message and receiving an acknowledgment). Note that ping messages and subscription messages have an almost identical latency in the GEO run; NoGEO has no ping latency as no ping messages are sent. Furthermore, the connect message latency stops after 250 seconds as clients are started with a 1-second offset on each machine.

From Figure 9.4a, one can also see that our prototype scales well vertically: For 750 clients, the average publish latency is 101ms on t3.micro and 62ms on t3.xlarge. Here, GeoBroker has to deliver 23.8 million messages. For 1000 clients, the average publish latency is 293ms on t3.micro and 107ms on t3.xlarge. Here, GeoBroker has to deliver 48.7 million messages.

[34]On t3.xlarge, the figures look very similar even though absolute latency values are smaller.

(a) GEO (b) NoGEO

Figure 9.6: Publish latency for 250 clients on t3.micro.



(a) GEO (b) NoGEO

Figure 9.7: Publish latency for 500 clients on t3.micro.

9.4 Summary

In summary, the results show that our prototype is sufficiently efficient and scales well vertically. In addition, we demonstrated that our approach can help to significantly reduce excess data dissemination for scenarios where geo-context matters; this preserves bandwidth and computational resources of subscribers and GeoBroker alike. While this comes with the cost of additional computation effort for the broker, we show that this overhead is relatively small at low load levels and is, at higher load levels, more than offset by the performance improvements gained by only transmitting relevant messages. While the absolute latency is not as low as the one of commercial services, this does not matter for evaluating the performance of the GeoBroker approach since GEO and NoGEO experiments were run with the same prototype.

Chapter 10

DisGB

In this chapter, we present the evaluation of DisGB. We start with experiments that we conducted on a small, distributed testbed with our proof-of-concept prototype (Section 10.1). The goal of these experiments is twofold. First, we want to show that using event and subscription geo-context information to select RPs is feasible in practice. Second, we want to experimentally validate our scenario analysis from Section 6.4.

Since experiments with multi-threaded distributed systems can never be fully deterministic, we decided not to use a similar approach for comparing our RP selection strategies with the state of the art. Instead, we designed a simulation study for that purpose (Section 10.2). We summarize our evaluation results in Section 10.3.

This chapter is based on material previously published in the Proceedings of UCC 2020 [73].

10.1 Experimental Validation of the Scenario Analysis

For the experiments, we set up three brokers in different AWS regions. Two brokers are near to each other (Paris and Frankfurt), and one is located further away (Columbus, Ohio). In each region, 1200 clients running on separate machines connect to their local broker and continuously send messages that resemble the scenario workloads from Section 6.4.2.

Each scenario workload comprises a set of client actions such as location updates, subscription updates (includes initial creation of subscriptions), and event publishings. See Table 10.1 which shows a high-level overview of the individual workload characteristics resulting from running each workload for 15 minutes. Geofence overlaps describe how many geofences overlap with more than one broker area; when there is no geofence ("–" in the table), an event or subscription must be distributed to all brokers.

Table 10.1: The characteristics of each workload depend on the underlying scenario.

Number of ...	Open Env.	Hiking	Ctx.-based
Location updates	7,200	193,709	69,236
Subscription updates	75,168	128,008	5,544
Subscription geofence overlaps	1,225	502	–
Event publishings	216,029	111,811	62,675
Event geofence overlaps	–	623	222

10.1.1 Experiment Results

A key characteristic when comparing pub/sub routing strategies is the amount of inter-broker messages, i.e., location updates, subscription updates, and published events, that are distributed by an LB of a client to RPs. Fewer inter-broker messages indicate a more efficient routing strategy. Figures 10.1 to 10.3 show the number of inter-broker messages for each strategy and scenario. The DisGB values are derived directly from the experiments. As a baseline, we could not use DisGB in single mode, as a centralized broker then processes all messages; here, no routing is necessary. Instead, we calculated how flooding events (FLE) or flooding subscriptions (FLS) would perform (see Section 2.3.3). We chose these strategies as a baseline, as flooding events is very similar to selecting RPs close to the subscriber, and flooding subscriptions is very similar to selecting RPs close to the publishers. Similar to our strategies, both flooding strategies also result in the lowest possible latency as events/subscriptions are sent directly to all brokers that might need them for matching. The difference is that our strategies use the event or subscription geofence to limit the number of RPs, which reduces the number of inter-broker messages. We present a more extensive comparison with approaches from related work in Section 10.2.

When selecting RPs close to the subscribers, the only inter-broker messages are events that need to be distributed to other brokers. Based on our theoretical assessment from Section 6.4.1, events must be distribted to $|E|$ brokers, i.e., all brokers whose broker area intersects with the event's geofence. Thus, the numbers in Figures 10.1 to 10.3 are as expected from our scenario analysis (see Section 6.4.1) because the number of inter-broker messages equals the number of event geofence overlaps (there are no overlaps between Columbus and Paris or Columbus and Frankfurt).

When flooding events, again the only inter-broker messages are events that need to be distributed to other brokers. Each published event must be distributed to all other brokers. For example, this leads to 223,622 (=2$*$111,811) inter-broker messages for the Hiking scenario. Note that the real inter-broker messages from the DisGB experiment match the calculated FLE messages in the Open Environmental Data scenario. This makes sense, as there are no event geofences and DisGB can, thus, not limit the number of target brokers.

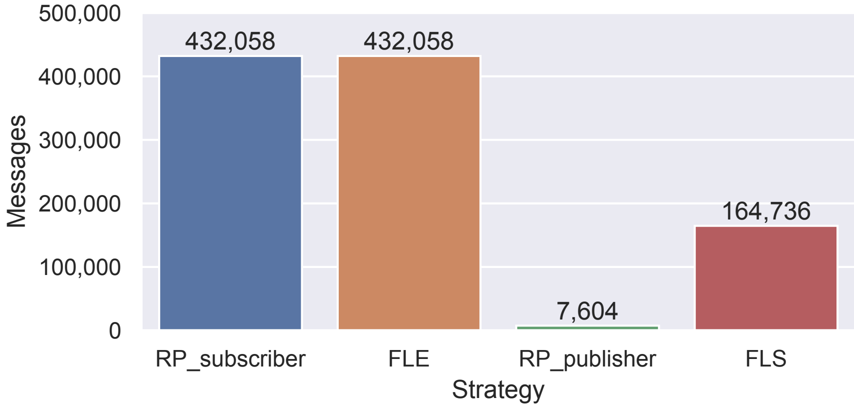


Figure 10.1: Open environmental data: selecting RPs close to the publisher is the most efficient strategy.

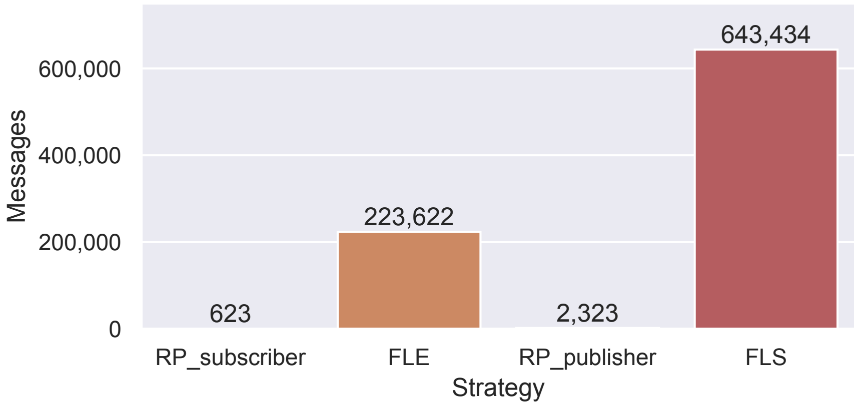


Figure 10.2: Hiking: both RP selection strategies are more efficient than the flooding strategies.

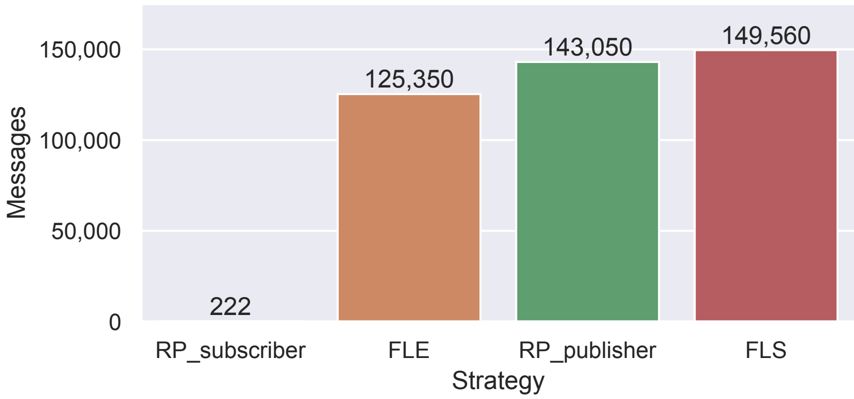


Figure 10.3: Context-based data distribution: selecting RPs close to the subscribers is the most efficient strategy.

When selecting RPs close to the publishers, inter-broker messages contain subscriber location updates, subscription updates, and events that have been successfully matched when the RP is not the LB. Subscriber location updates need to be distributed to $|\bigcup_{i=1}^{n} S_i|$ brokers; it is not trivial to calculate this number. The only exception is the context-based data distribution scenario, for which no subscription geofence exists. Here, every subscription update and location update is sent to the two other brokers: 149,560 (=2∗69,236 + 2∗5,544). However, DisGB does not distribute the location updates of a subscriber if the subscriber has not created a subscription yet; thus, the actual number is lower (143,050). Furthermore, this number also includes events that have been successfully matched against a subscription created by a client connected to another broker. In the open environmental data scenario and in the hiking scenario, there are only few subscription geofence overlaps. Thus, the number of inter-broker messages is significantly lower.

When flooding subscriptions, every subscription or subscriber location update must be distributed to all other brokers. In the Hiking scenario, this leads to 643,434 (=2∗193,709 + 2∗128,008) inter-broker messages. In addition, successfully matched events of subscribers connected to another broker must also be distributed. Successful matches can only be determined through simulation or an experiment, so the calculated number of inter-broker messages reported in the figures is slightly lower than FLS would show in practice.

10.1.2 Implications

With our experiments, we demonstrate that using geo-context information to select RPs is feasible in practice. We also validated our scenario analysis from Section 6.4.2: For scenarios without or with very large event geofences (Open Environmental Data scenario), selecting RPs close to the publisher is the better strategy, as this reduced inter-broker messages by 98.2% (7,604 inter-broker messages compared to 432,058). For scenarios with similar event and subscription geofences (Hiking scenario), none of the two strategies has a clear advantage over the other one (2,323 inter-broker messages compared to 623), while both performed significantly better than their respective baselines. For scenarios without or with very large subscription geofences (Context-based Data Distribution scenario), selecting RPs close to the subscribers is the better strategy, as this reduced inter-broker messages by 99.8% (222 inter-broker messages compared to 143,050).

10.2 Simulation-based Comparison to the State-of-the-Art

In this section, we describe the results of comparing our proposed RP selection strategies to related work. For this, we use simulation (Section 10.2.1) with a newly developed simulation tool (Section 10.2.2) that allows us to compare different routing strategies (Section 10.2.3) in a fully deterministic way. Our results (Section 10.2.4) indicate that our strategies reduce the event delivery latency by up to 22 times while still requiring less inter-broker messages than most other strategies.

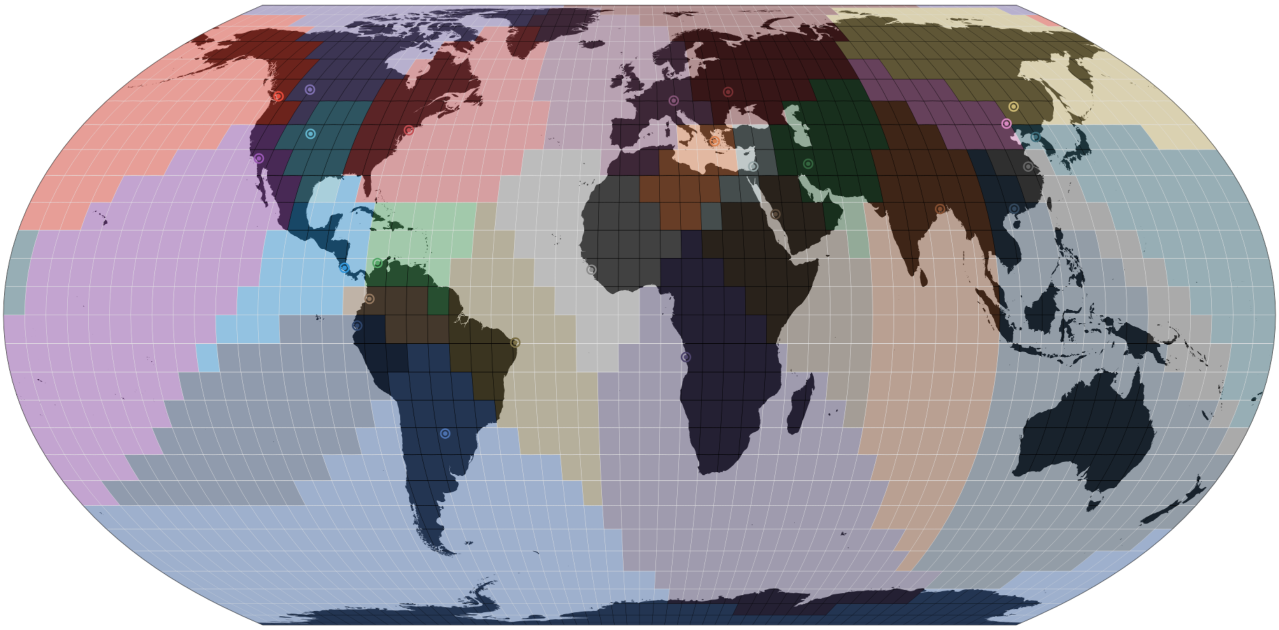


Figure 10.4: Broker locations and broker areas for a simulation with 25 brokers.

10.2.1 Simulation Design

Our simulation model follows a realistic setup: brokers are distributed across the globe and clients connect to their respectively nearest broker. For each simulation run, based on a given total number of brokers and clients, we determine broker and client locations as follows:

1. Place brokers at the locations of randomly chosen cities with a population of more than 1 million citizens based on the world cities dataset [175].
2. Assign clients in proportion to the population of a broker's city and the total population of all chosen cities to each broker.
3. Generate random locations for each client of a broker in the area that is closest to this broker[35]; this area is also the broker area needed for the DisGB strategies.

Figure 10.4 visualizes broker locations and broker areas for 25 brokers. We also developed a tool for interactively exploring simulation setups, a live demo is available on GitHub[36].

In total, we compare seven RP selection strategies (Section 10.2.3). Our simulation workload is based on the Hiking scenario from the scenario analysis and experiments, as for this neither of our two strategies has a clear advantage. While the distribution of subscriptions and events between brokers is different for each strategy, the result from the client perspective does not change: each client receives the same events based on its individual subscriptions regardless

[35]We determine this area by dividing the earth surface into non-overlapping squares and then assigning each square to the broker whose location is closest to the square's center. Other methods such as Voronoi diagrams could also be used.

[36]https://moewex.github.io/DisGB-Simulation/

of the RP selection strategy used. For this, all brokers have to do a ContentCheck and the two GeoChecks when matching messages (see Section 5.2). Otherwise, for example, the other strategies would also deliver events to subscribers that are not present in the event's geofence. While each RP selection strategy has its own way of distributing events, subscriptions, or both to the corresponding RPs, not all can appropriately distribute client locations. Still, client locations are needed for the two GeoChecks. Enhancing the strategies to distribute location updates is out of scope for this thesis. Thus, we assume that each broker floods client location updates to all other brokers for all simulation runs, even though our proposed RP selection strategies are more efficient than this. Due to this decision, we disregard the effects of location updates for the analysis of simulation results (Section 10.2.4).

We decided to have 1000 distinct topics, each client subscribed and published to five topics. Furthermore, we used circular geofences (arbitrary shapes are possible in practice); all event geofences had a radius of 5 degree (about 555km at the equator) and all subscription geofences had a radius of 10 degree (about 1110km at the equator). We decided to use such large values for the geofences to increase the amount of inter-broker traffic; smaller geofences particularly benefit DisGB. For each RP selection strategy, we ran one simulation round with 100,000 clients and 25 brokers[37], one with 1,000,000 clients and 25 brokers, and one with 100,000 clients and 256 brokers[38].

With the simulation, we can discuss how each RP selection strategy affects three aspects: First, the number of inter-broker event and subscribe messages; fewer messages means that the strategy is more efficient. Second, the event delivery latency: how much time does it take for a client to receive an event after it has been published by any other client connected to any of the brokers. Third, the subscription update delay: how much time does it take for a broker to receive a subscription update distributed by another broker. In general, having a lower event delivery latency and subscription update delay means that a strategy offers superior performance compared to a strategy with higher values.

10.2.2 Simulation Tool

The simulation tool can be customized with several parameters. The six most important parameters are: the chosen RP selection strategy, the number of brokers, clients, and distinct topics, the size of event geofences, and the size of subscription geofences. The simulation tool is fully deterministic, i.e., repeating a simulation run with the same parameters (and random seed) leads to the same result. Our tool implements a parallel discrete-event simulation [111]. The simulation time granularity is 1 ms; we refer to one time unit as a *tick*.

Upon startup, our simulation tool generates a workload based on its input parameters. Every workload comprises a set of brokers that use the specified RP selection strategy and a set of

[37] We chose this number based on the number of active AWS Cloud regions (24 at that time).
[38] The GQPS strategy (Section 10.2.3) works best if the square root of the number of brokers is an integer.

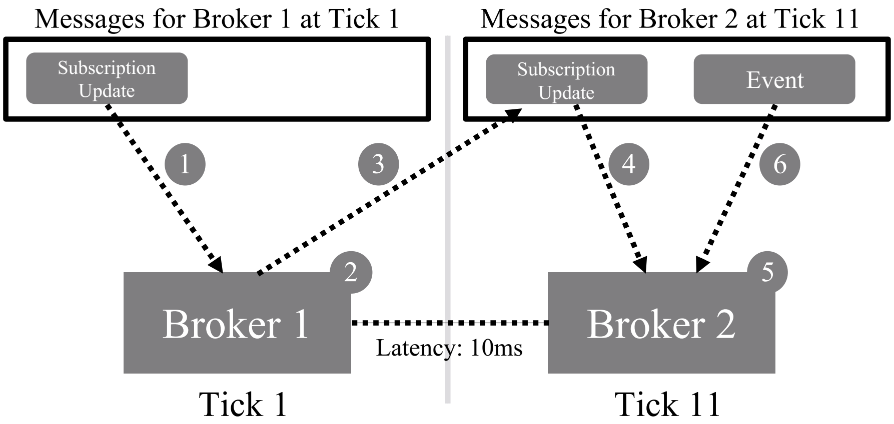


Figure 10.5: Broker 1 receives a subscription update at Tick 1 (1). Besides updating its state (2), it must distribute the update to Broker 2 as specified by its RP selection strategy (3). As the latency is 10 ms, Broker 2 receives the subscription update at Tick 11 (4). It does not need to be further distributed, so it is only used to update the broker's state (5). After all brokers have finished processing subscription updates, Broker 2 can continue to process the next type of message (6).

client messages that should be received by an LB at a certain tick. After the initialization, the simulation tool sequentially processes ticks and the corresponding messages. During each tick, the tick's messages are delivered in a pre-defined order to brokers, which use them to update their state, e.g., managed client locations or subscriptions, accordingly. Besides state updates, brokers might also further distribute messages to other brokers or deliver events to clients. For this, they determine when a message would be received by their target and then add this message to the event queue for the corresponding tick (see Figure 10.5).

The latency between two brokers is calculated by multiplying their physical distance with the constant 0.021 ms/km. We determined this constant by analyzing the 2016 IPlane traceroute dataset [116]. For communication between a client and its LB, we use a fixed latency of 5 ms so that we can determine which part of the overall latency is caused by inter-broker traffic.

Especially for larger client and broker numbers, storing the full set of sent and received messages for analysis is not feasible, e.g., for a single run we experienced up to 200,000,000 messages. Thus, we calculated all result values presented in Section 10.2.4 by analyzing the stream of data. We used the P2 algorithm by Jain and Chlamtac [93] to compute percentiles heuristically. The mean squared error of the P2 algorithm "is comparable to that obtained by order statistics and [...] both tend to zero as the sample size is increased" [93]. The simulation tool is implemented in Kotlin and available as open-source on GitHub[39].

[39] https://github.com/MoeweX/DisGB-Simulation

10.2.3 RP Selection Strategies

In our simulation tool, we have implemented our two RP selection strategies (Section 6) and five additional strategies from related work, including our BCGroups strategy from Chapter 4. In the following, we briefly describe at which brokers the matching occurs for these five additional strategies (i.e., how the RPs are determined). We chose these strategies because they are either well known or because our two strategies do not obviously offer superior performance when geo-context information is available.

Flooding Events (Flood_E): Every broker is an RP for every event. When an LB of a publisher receives an event, it distributes it to all other brokers. After matching this event, brokers can deliver the event to their local subscribers directly.

Flooding Subscriptions (Flood_S): The LB of a publisher is the only RP for a given event. When an LB of a subscriber receives a subscription update, it distributes it to all other brokers. After matching an incoming event, the LB of the publisher might have to distribute the event to the LBs of matching subscribers. This is necessary, as each client only communicates with its LB.

Consistent Hashing (DHT): This strategy builds on distributed hash tables and is used in pub/sub systems such as Scribe [147] or Hermes [134]. The RP is determined by mapping the event and subscription topics to a particular broker with consistent hashing [100]. Once an RP has matched the event, it notifies the LBs of matching subscribers about successful matches so that they can deliver the event to their local subscribers.

Grid Quorum (GQPS): To determine RPs, an application-level overlay network is created that makes each broker addressable by a position in a grid, i.e., by its row and column. RPs are all brokers in the same row or column as the LB, so this is where events and subscriptions must be sent [177]. After matching an event, the RP notifies the LBs of matching subscribers that have not been in the same row/column as the publisher's LB about successful matches.

BCGroups (BG): Physically close brokers create broadcast groups in which events are flooded to other group members for matching, i.e., all group members of a publisher's LB are an RP. Furthermore, one broker of each broadcast group (the leader) aggregates and forwards events and subscriptions originating in its group to a centralized cloud broker[40]. The cloud broker matches events with subscriptions created by the leaders; therefore, it is also an RP. If an event is matched successfully at the cloud broker, the corresponding leader and all its members become an RP for this event to match and deliver it [82].

In the following, the abbreviation for selecting RPs close to the subscribers is DisGB_E, as with this strategy events are distributed. Similarly, the abbreviation for selecting RPs close to the publishers is DisGB_S, as with this strategy subscriptions are distributed.

[40] AWS operates its biggest data center in West Virginia, USA, so that is where we put the cloud broker for the simulation.

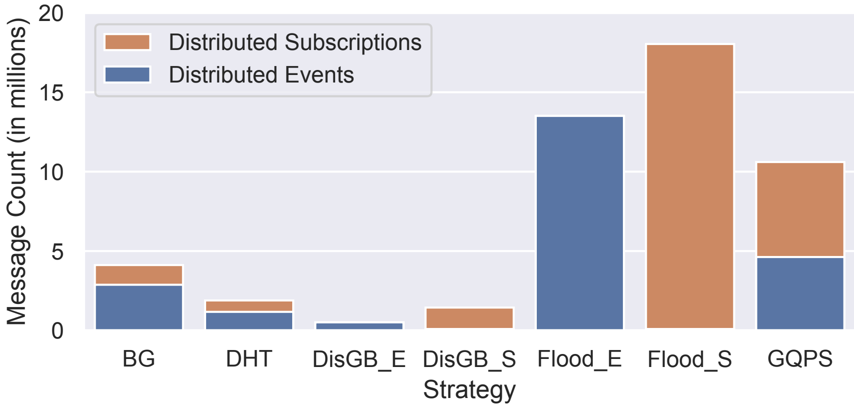


Figure 10.6: For 100,000 clients and 25 brokers, both DisGB strategies require less inter-broker messages than all other strategies.

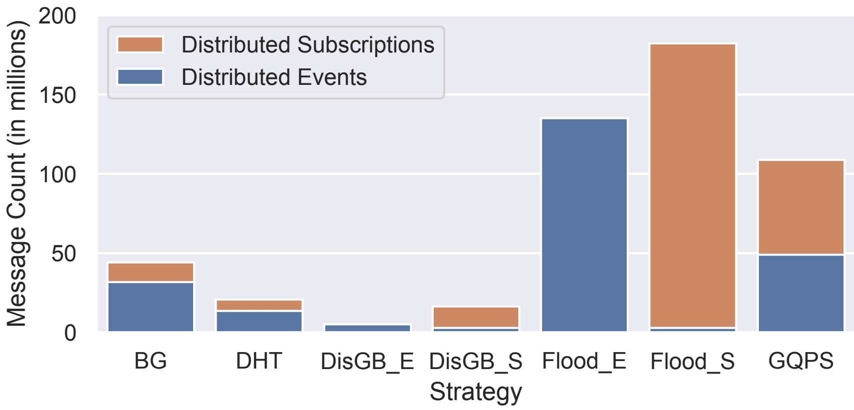


Figure 10.7: For 1,000,000 clients and 25 brokers, the result is very similar to the one shown in Figure 10.6.

10.2.4 Simulation Results

In total, we ran 27 simulation rounds that each comprised a fifteen minute workload in simulation time[41]. For all strategies, we do not take preparatory actions, e.g., running the broadcast group formation process, into account for the simulation results since their impact is constant as long as the runtime infrastructure does no change.

Figure 10.6 shows the number of distributed inter-broker event or subscription messages for 100,000 clients and 25 brokers. The first notable observation is that **both DisGB strategies require less inter-broker messages than all other strategies**. When observing the two flooding strategies, one can see that Flood_S distributes more subscriptions than Flood_E distributes events. Therefore, clients update subscriptions more often than they publish events in the workload used. This, and the fact that the subscription geofence radius is twice the radius of event geofences, also explains why the DisGB_E strategy requires less inter-broker messages

[41] The execution time was up to 16 hours for a single simulation run.

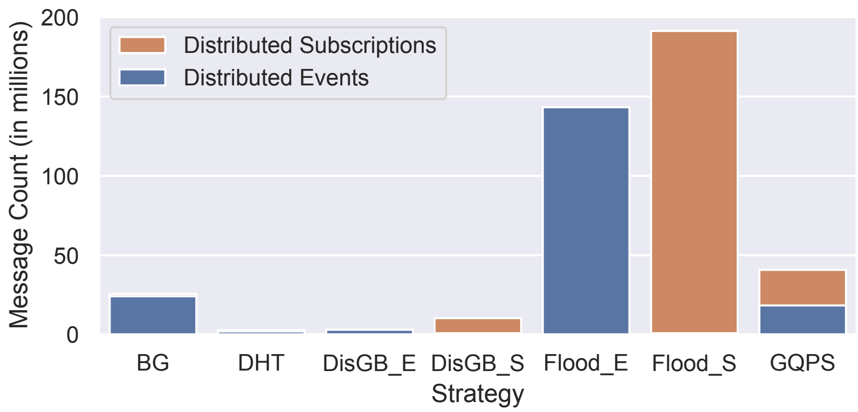


Figure 10.8: For 100,000 clients and 256 brokers, DHT requires the fewest inter-broker messages.

than DisGB_S. Using 10 times more clients (1,000,000) only has the effect that each strategy requires about 10 times more inter-broker messages (see Figure 10.7). Using approximately ten times more brokers (256), however, does reduce the amount of inter-broker messages for most strategies compared to the two flooding strategies (see Figure 10.8). Furthermore, the DHT strategy now requires the lowest amount of messages because exactly one broker is responsible for matching events and subscriptions of a given topic.

Figure 10.9 shows the event delivery latency and the subscription update delay for 100,000 clients and 25 brokers. As the subscription update delay describes how much time it takes for a broker to receive a subscription update distributed by another broker, Flood_E and DisGB_E do not have such a delay. With all other strategies, a subscriber might still receive events to which it has already unsubscribed at its LB. The minimum event delivery latency is 10 ms; this latency can only be achieved if the publisher and subscriber are connected to the same broker. One can see that this is the case for more than 75% of the delivered events, since the first three quartiles of the event delivery latency for BG, DisGB_E, DisGB_S, Flood_E, Flood_S, and GQPS are at 10 ms. Still, comparing the mean and standard deviation of the strategies (Table 10.2) reveals that only **both DisGB strategies offer the same event delivery latency as the two flooding strategies**.

Furthermore, the event delivery latency of both DisGB strategies is up to 22x lower than the one achievable by strategies found in related work. Using 10 times more clients (1,000,000) does not significantly influence the event delivery latency or subscription update delay (see Figure 10.10). Using ten times more brokers (256), however, does influence both values (see Figure 10.11) as subscribers are more often connected to a different broker than the publisher of a matching event (the third quartile is not at 10 ms anymore). Still, both DisGB strategies continue to offer a similar event delivery latency as their respective flooding counterparts (while requiring significantly less inter-broker messages).

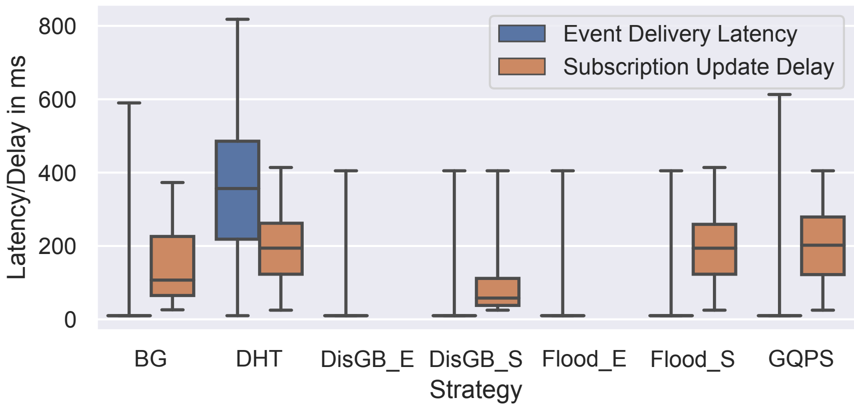


Figure 10.9: For 100,000 clients and 25 brokers, both DisGB strategies offer a similar performance as their respective flooding counterparts.

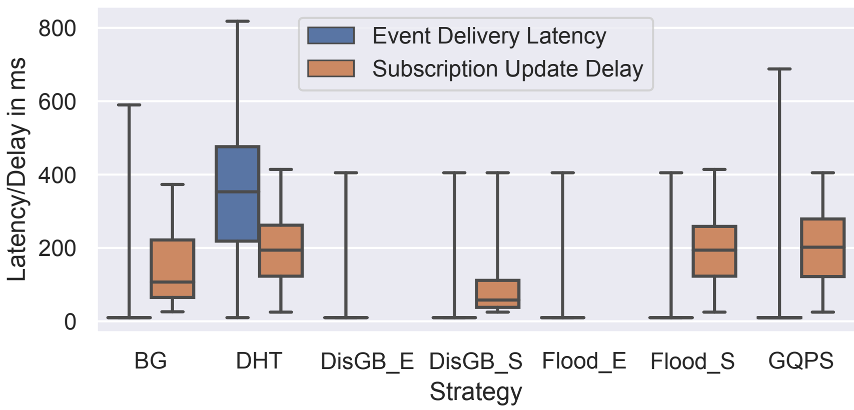


Figure 10.10: For 1,000,000 clients and 25 brokers, the result is very similar to the one shown in Figure 10.9.

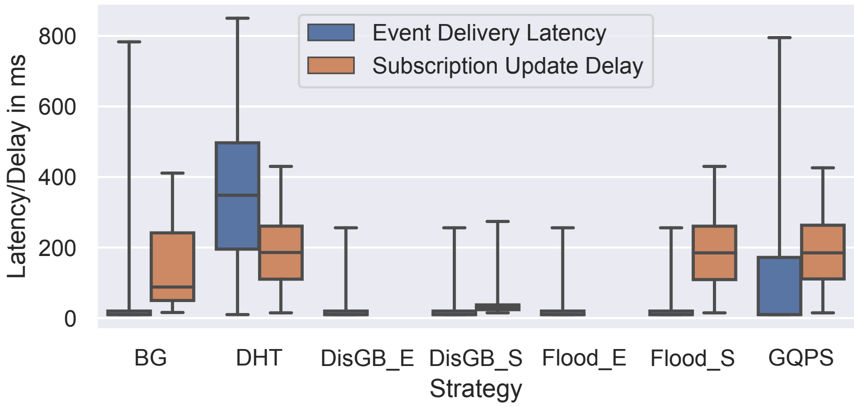


Figure 10.11: For 100,000 clients and 256 brokers, latency and delay increase slightly compared to Figure 10.9 as having more brokers increases the likelihood of inter-broker communication.

Table 10.2: Average event delivery latency and standard deviation (in brackets) in ms for different broker (B.) and client (C.) numbers.

Strategy	25 B., 100k C.	25 B., 1M C.	256B. 100k C.
BG	22 (63)	27 (75)	24 (69)
DHT	353 (188)	348 (184)	355 (195)
DisGB_E	16 (26)	19 (31)	16 (8)
DisGB_S	16 (26)	19 (31)	16 (8)
Flood_E	16 (26)	19 (31)	16 (8)
Flood_S	16 (26)	19 (31)	16 (8)
GQPS	31 (85)	39 (98)	105 (145)

10.3 Summary

In summary, the experiments with our proof-of-concept prototype demonstrate that using geo-context information to select RPs is feasible in practice. Furthermore, we validated our scenario analysis from Section 6.4: We confirmed that selecting RPs close to the publisher should be used for scenarios with very large event geofences and small subscription geofences. Selecting RPs close to the subscribers should be used for scenarios with very large subscription geofences and small event geofences.

With the simulation analysis, we showed that both DisGB strategies require less inter-broker messages than all other strategies when there are few brokers (25). If there are many brokers (256), only the DHT strategy requires less inter-broker messages. At the same time, however, the event delivery latency of both DisGB strategies is up to 22x lower than the one achievable by DHT. Independent of the number of brokers or clients, both DisGB strategies continue to offer a similar event delivery latency as their respective flooding counterparts while requiring significantly less inter-broker messages.

Chapter 11

MockFog

In this chapter, we present the evaluation MockFog. After having shown with our proof-of-concept implementation MockFog 2.0 that this approach can indeed be implemented, we now use an example application to showcase its key features. For this, we run experiments with a fog-based smart factory application (Section 11.1) for which we emulate a runtime infrastructure with MockFog 2.0. In the experiments, we use an orchestration schedule (Section 11.2) that includes multiple infrastructure and workload changes and study the effects on the application (Section 11.3). We summarize our evaluation results in Section 11.4.

Note that our goal is to provide an overview of the features of MockFog 2.0 and not to design a realistic benchmark or system test for our example application. This evaluation also serves the purpose of showing how simple it is to run such application experiments with MockFog: one only needs to create configuration files[42] for the three modules. Then, MockFog sets up an infrastructure testbed in the cloud, handles the application roll-out and experiment orchestration, collects results (i.e., application output such as log files and the log files of node agents), and finally destroys the testbed. As a scenario, we build upon the smart factory application introduced in [132][43].

This chapter is based on material previously published in our MockFog 2.0 paper that is still under review [79].

11.1 Overview of the Smart Factory Example

In the smart factory, a production machine produces goods that are packaged by another machine. Based on input from a camera and a temperature sensor, the production rate and

[42]The configuration files of our experiments are available in the *node-manager/run-example-smartfactory* directory of our code repository: `https://github.com/MoeweX/MockFog2`

[43]The scenario application source code is available at `https://github.com/OpenFogStack/smart-factory-fog-example/tree/mockfog2`.

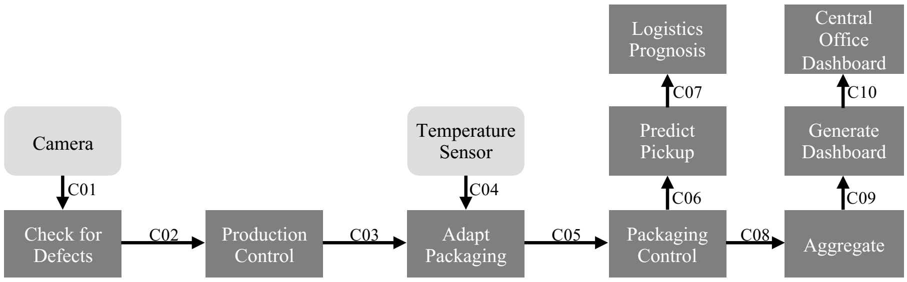


Figure 11.1: The smart factory application comprises 11 components and 10 communication paths between individual components (C01 — C10).

packaging rate are adjusted in real-time. The packaging rate is used to create a logistic prognosis, i.e., for scheduling the collection and delivery of goods. Furthermore, a dashboard provides a historic packaging rate overview.

Each of the components of the smart factory application communicates with at least one other component (see Figure 11.1). *Camera* sends its recordings to *check for defects* which notifies *production control* about products that should be discarded. Based on input from *production control* and *temperature sensor*, *adapt packaging* transmits the target packaging rate to *packaging control*. *Adapt packaging* calculates the packaging rate based on the current production rate, the backlog of produced but not packaged items, and the temperature input: packaging must be halted if the current temperature exceeds a threshold. *Packaging control* sends the current rate and backlog to *predict pickup* and *aggregate*. *Predict pickup* predicts when the next batch of goods is ready for pickup and sends this information to *logistics prognosis*. *Aggregate* aggregates multiple rate and backlog values to preserve bandwidth and transmits the results to *generate dashboard*. *Generate dashboard* stores the data in a database, creates an executive summary, and sends it to *central office dashboard*.

The smart factory application comprises components that react to events from the physical world (light grey boxes) and components that only react to messages received from other application components (dark grey boxes). For example, the *temperature sensor* measures the physical machine's operation temperature that packages goods. *Adapt packaging*, on the other hand, receives messages from other application components and has no direct interaction with the physical world.

When testing real-time systems, two important concepts are reproducibility and controllability [1, p. 263]. During experiments, *camera* and *temperature sensor* hence generate an input sequence that can be controlled by MockFog 2.0 to achieve reproducibility. For this, it is necessary to mock sensors in software. Furthermore, components that do something in the physical world based on received messages, e.g., *packaging control*, only log their actions when doing experiments rather than sending instructions to physical machines.

Figure 11.2 shows the machines and network links of the smart factory infrastructure. A gateway connects the camera, production machine, packaging machine, and temperature sensor.

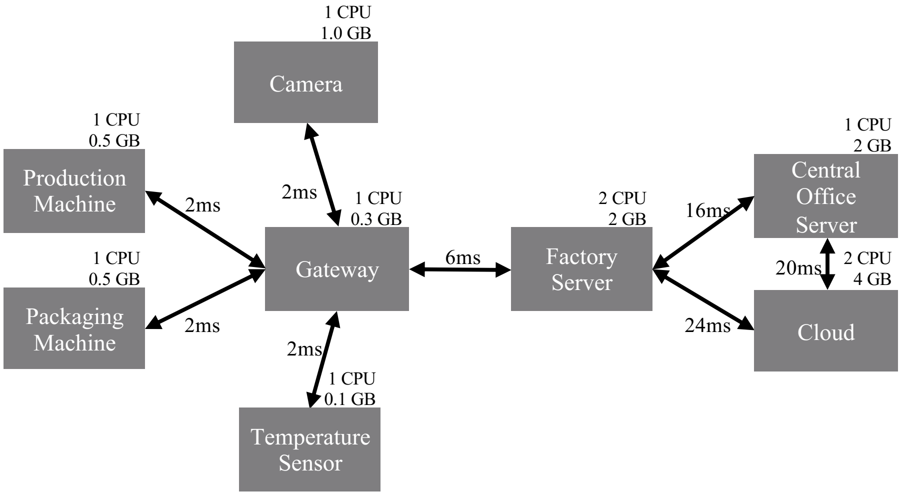


Figure 11.2: The smart factory infrastructure comprises multiple machines with different CPU and memory resources. Communication between directly connected machines incurs a round-trip latency between 2 ms and 24 ms.

Table 11.1: Mapping of application components to machines.

Application Component	Machine
Camera	Camera
Temperature Sensor	Temperature Sensor
Check for Defects	Gateway
Adapt Packaging	Gateway
Production Control	Production Machine
Packaging Control	Packaging Machine
Predict Pickup	Factory Server
Logistics Prognosis	Factory Server
Aggregate	Factory Server
Generate Dashboard	Cloud
Central Office Dashboard	Central Office Server

Each has a 2 ms round-trip latency to the gateway, 1 CPU core, and 0.1 GB to 1.0 GB of available memory. The gateway is connected to the factory server, which is connected to the central office server and the cloud. The cloud and central office server are also connected directly. All connections between machines are set up with a bandwidth of 1 GBit without any package loss, corruption, reordering, or duplicates. Table 11.1 shows the mapping of application components to machines. To derive such a mapping and to compare it to other approaches, developers can use approaches such as [84, 104, 172].

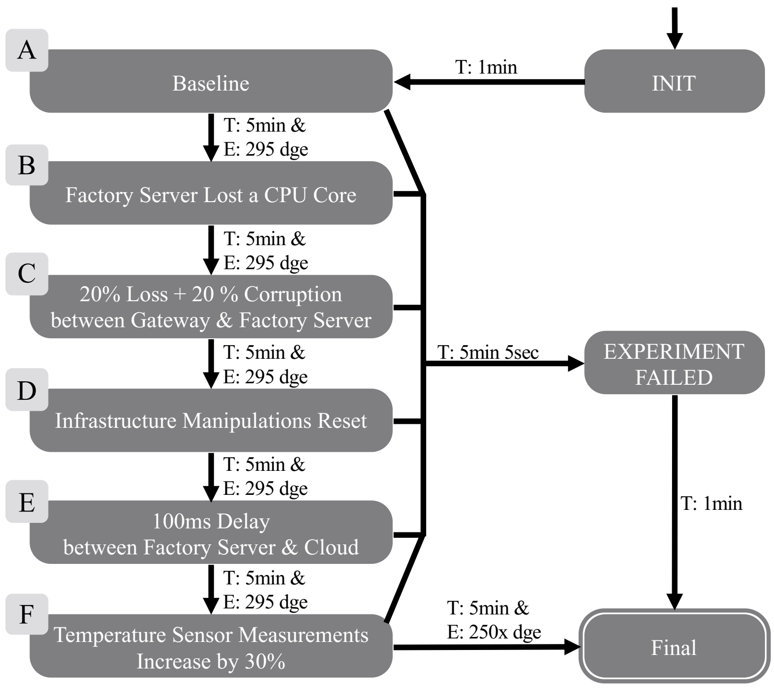


Figure 11.3: The orchestration schedule has nine states. During successful executions, the transitioning conditions mostly use a combination of time-based (5 minutes) and event-based conditions (receipt of 295 dashboard generated events (dge)).

11.2 Orchestration Schedule

For the experiments, we use an orchestration schedule with nine states (Figure 11.3). At the beginning of each state, MockFog 2.0 instructs *camera* and *temperature sensor* to restart their workload data sequence. Thus, the application workload is comparable during each state. The schedule starts with *INIT*; after a minute, MockFog 2.0 transitions to state *A*. The purpose of state *A* is to establish a baseline by running the application in an environment that closely mimics the real production environment. At runtime, *generate dashboard* creates a new dashboard once per second and sends a notification to the node manager. We use this event as a failure indicator in all states; if it has not been received at least 295 times within a timeframe of 5 minutes, the experiment failed. If, however, it has been received 295 times and five minutes have passed, MockFog 2.0 transitions to the next state.

For state *B*, MockFog 2.0 changes the infrastructure: the factory server has only access to one CPU core instead of two. Then, in state *C*, loss and corruption on the network link between gateway and factory server are set to 20%. Note that the factory server has not regained access to its second CPU core. In state *D*, all infrastructure changes are reset; the environment now again closely mimics the real production environment and the application can stabilize. In state *E*, the round-trip latency for messages sent from the factory server to the cloud is increased from 24 ms to 100 ms. In state *F*, the latency is reset to 24 ms but the temperature sensor is instructed to change the measurement generation: the average temperature sensor measurements are now 30% higher, which causes the packaging machine to pause more frequently. This, in turn, should decrease the average packaging rate and increase the average packaging backlog. After state *F*, MockFog 2.0 transitions to *FINAL* and the experiment orchestration ends. If at any point a failure occurs, MockFog 2.0 will transition to *EXPERIMENT FAILED*.

11.3 Results

In the following, we first validate that running the orchestration schedule leads to reproducible results (Section 11.3.1). Then, we analyze how the changes made in each state of the orchestration schedule affect the smart factory application (Section 11.3.2) and summarize our results. Note that the analysis of application logs and other output files is not done by MockFog since this is entirely application-specific and also depends on the load generator used. Interpreting which latency, processing time, respective variance values, etc. are acceptable, also depends on the concrete application. Thus, our results and conclusions are only valid for our specific use case and primarily serve the purpose of demonstrating how this process could look like in practice.

11.3.1 Experiment Reproducibility

To analyze reproducibility, we repeat the experiment five times. For each experiment run, we bootstrap a new infrastructure, install the application containers, and start the experiment orchestration – this is done automatically by MockFog 2.0. After the experiment run, we calculate the average latency for each communication path (C01 to C10 in Figure 11.1). Ideally, the latency results from all five runs should be identical for each communication path; in the following, we refer to the five measurement values for a given communication path as latency set. In practice, however, it is not possible to achieve such a level of reproducibility because the application is influenced by outside factors [1, p. 263]. For example, running an application on cloud VMs and in Docker containers already leads to significant performance variation [63, 64]. To measure this variation, we use the median runs of each latency set as a baseline and calculate how much individual runs deviate from this baseline (see Figure 11.4).

Considering that we are running the experiments in a public cloud, we can see in the figure that the deviation is small for almost all communication paths. The exception are communication paths C06 and C07 in states *B*, *C*, and *E* which show the most variance across runs. In these states, the node manager applies various resource limits on the factory server. Reducing the available compute and network resources seems to impact the stability of affected communication paths negatively. Identifying such cases, however, in which infrastructure changes negatively impact application stability is exactly for what we designed MockFog. Thus, we can conclude that experiment orchestration leads to reproducible results under normal operating conditions. This holds true even if a new set of virtual machines is allocated for each run. When a higher experiment reproducibility is required than the one achievable in a public cloud, one could, for example, add support for private OpenStack clusters to the infrastructure module.

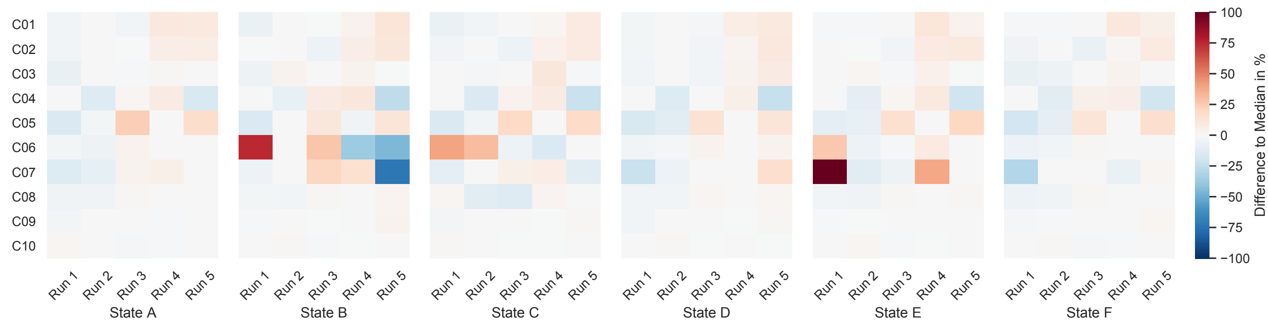


Figure 11.4: Latency deviation across experiment runs is small for most communication paths even though experiments were run in the cloud. On paths C06 and C07, resource utilization is high in states *B*, *C*, and *E* leading to the expected variance across experiment runs.

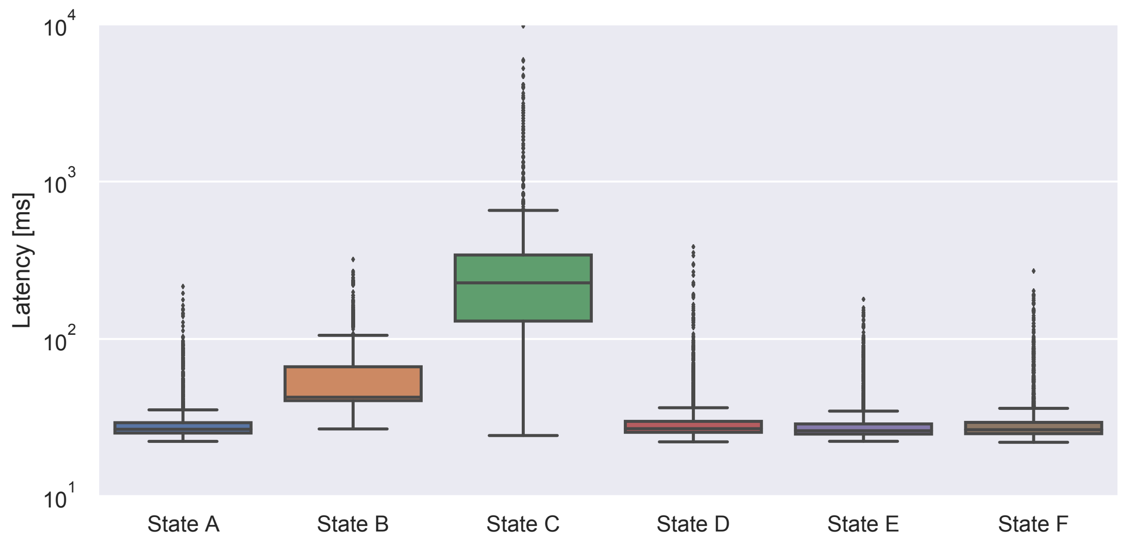


Figure 11.5: Latency between *packaging control* and *logistics prognosis* is affected by both CPU and network restrictions.

11.3.2 Application Impact of State Changes

Of the five experiment runs, the second run is the most representative for the orchestration schedule: On average, the latency of its communication paths deviate by 4.45% from the median latency of the set. Run three, four, five, and one deviate by 5.20%, 5.28%, 9.77% and 10.23% respectively. Thus, we select the second run as the basis for analyzing how the changes made in each state affect application metrics.

Figure 11.5 shows the latency between *packaging control* and *logistics prognosis*. This latency includes the communication path latency of C06 and C07, as well as the time *predict pickup* needs to create the prognosis. In states *A*, *D*, *E*, and *F*, there are either no infrastructure changes or the ones made are on alternative communication paths; thus, latency is almost identical. In state *B*, the factory server loses a CPU core; as a result, *predict pickup* needs more time to create a prognosis which increases the latency. In state *C*, the communication path C06

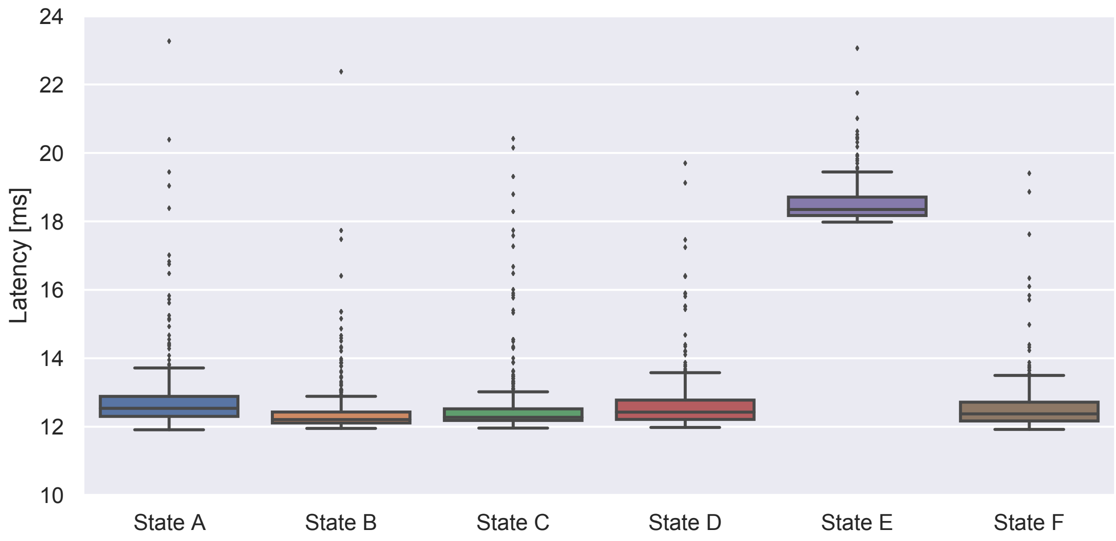


Figure 11.6: Latency on C09 between *aggregate* and *generate dashboard* is affected by the delay between factory server and cloud.

additionally suffers from a 20% probability of package loss and a 20% probability of package corruption. As these packages have to be resent[44], this significantly increases overall latency.

Figure 11.6 shows the latency on C09, i.e., the time between *aggregate* sending and *generate dashboard* receiving a message. In states *A*, *D*, and *F*, there are either no infrastructure changes or the ones made are on alternative communication paths; thus, latency is almost identical. Note that the minimum latency is 12 ms; this makes sense as the round-trip latency between factory server and cloud is 24 ms. In states *B* and *C*, the factory server loses a CPU core; MockFog 2.0 implements this limitation by setting Docker resource limits. As a result, there is now 1 CPU core that is not used by the application containers and hence available to the operating system. As the resource limitation seems not to impact *aggregate*, the additional operating system resources slightly decrease latency. While the effect here is only marginal, one has to keep such side effects in mind when doing experiments with Docker containers. In state E, the round-trip latency between factory server and cloud is increased to 100 ms. Still, the minimum (one-way) latency only increases to 18 ms as packages are now routed via the central office server (round-trip latency is 16 ms + 20 ms).

The *packaging control* reports its current packaging rate once a second. Figure 11.7 shows the distribution of reported values, i.e., how often each packaging rate was reported per state. In states *A*, *B*, *C*, *D*, and *E*, the workload generated by *camera* and *temperature sensor* is constant, so the rates are similar. In state *F*, however, the temperature sensor distributes measurements that are 30% higher on average. As a result, the packaging machine must halt production more frequently, i.e., the packaging rate equals zero. This also increases the backlog so the packaging machine will more frequently run at full speed to catch up, i.e., the packaging rate equals 15.

[44]Resent packages can also be impacted by loss or corruption.

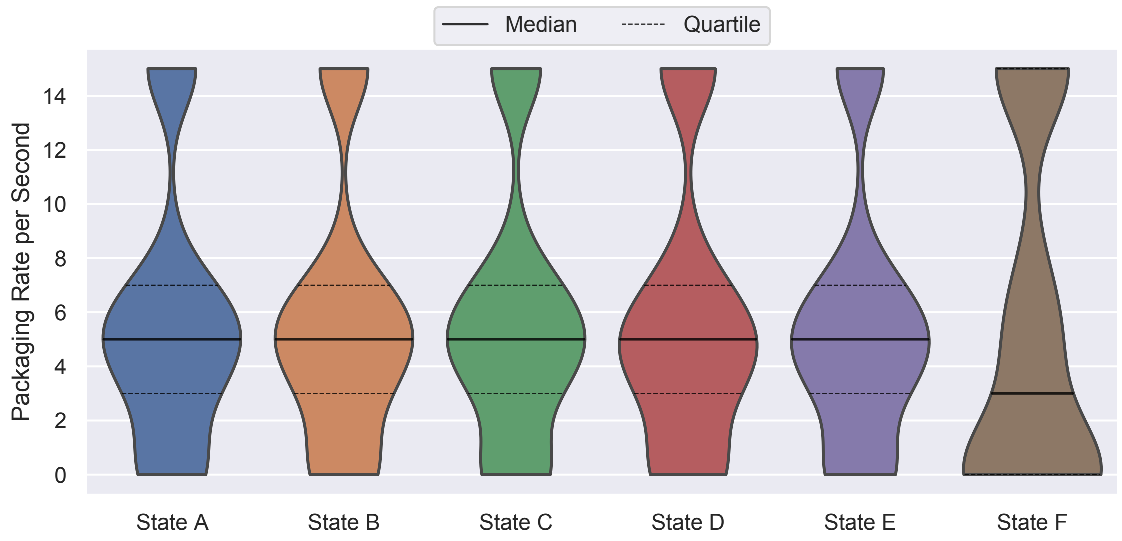


Figure 11.7: Distribution of packaging rate per state: When the temperature increases in state *F*, *packaging control* needs to pause more often resulting in more frequent packaging rates of 0 (machine is paused, 1st Quartile) and 15 (machine is running at full speed to catch up on the backlog, 3rd Quartile).

11.4 Summary

In summary, the experiments show that MockFog 2.0 can be used to automatically set up an emulated fog infrastructure, install application components, and orchestrate reproducible experiments. As desired, changes to infrastructure and workload generation are clearly visible in the experiment results. The main benefit of the MockFog approach is that this autonomous process can be integrated into a typical application engineering process. This allows developers to automatically evaluate how a fog application copes with a variety of infrastructure changes, failures, and workload variations after each commit without access to a physical fog infrastructure, with little manual effort, and in a repeatable way [18].

Part IV

Conclusions

This final part shall conclude this thesis. For this purpose, we start in Chapter 12 with a summary of the two main needs for building, testing, and evaluating fog-based pub/sub systems and our three main contributions BCGroups, (distributed) GeoBroker, and MockFog. In Chapter 13, we discuss general limitations of these contributions and potential future research directions.

Chapter 12

Summary

Distributing IoT data via fog-based pub/sub systems has many advantages. This includes low latency communication between physically close clients, better availability, and reduced bandwidth consumption in the wide-area network. For building, testing, and evaluating such fog-based pub/sub systems, we identified two main needs:

First, the need for considering the unique characteristics of the fog and the IoT. When distributing data across the fog, there is a tradeoff between latency and excess data dissemination. Existing solutions do either not address the tradeoff or require a holistic view on all events and subscriptions. Moreover, while many state of the art solutions are very efficient in terms of required inter-broker messages, they might route messages across any node even though the node might not have access to sufficient compute or networking resources. Particularly in the fog, this is not acceptable due to the heterogeneity of machines, as well as partly unstable and relatively slow network connections. Furthermore, a unique characteristic of the IoT is that data generated by IoT devices is often only relevant to physically close devices or devices in a specific physical area. Today's pub/sub systems, however, mostly focus on data content and not on the associated geo-context when deciding what data should be distributed to which clients. Using such IoT-specific domain knowledge can significantly increase efficiency while still providing low latency communication to clients.

Second, the need for automated execution of fog application experiments in a controllable environment: As the fog comprises highly distributed machine resources, setting up testing infrastructure and managing application components is more complex than the ease of adoption developers are used to from the cloud. Furthermore, especially during early development stages, a fog infrastructure might not even exist yet. Thus, evaluating fog applications and running experiments with fog systems is difficult, especially when aiming for a high reproducibility and controllability level.

To address these needs, we presented three main contributions in this thesis:

1. *BCGroups: An Inter-Broker Routing Strategy for the Fog.* Here, the main idea is to split the set of fog brokers into well connected broadcast groups which use flooding for intra-group communication and a cloud RP for inter-group communication. This minimizes the communication latency of client devices in physical proximity, i.e., where a low communication latency is often required in IoT scenarios. With the broadcast group size, we can control the number of flooded events and hence manage the tradeoff between excess data and latency.
2. *GeoBroker & DisGB: Leveraging Geo-Context for IoT Data Distribution.* These systems leverage the full geo-context of publishers and subscribers to reduce excess data dissemination. GeoBroker is designed to run on a single machine and reduces the machine's load, bandwidth consumption, and the number of irrelevant messages that need to be processed by subscribers. DisGB additionally introduces two novel strategies for routing messages between brokers. While the strategies can only be used for scenarios where geo-context matters, using them reduces excess data dissemination even beyond what is possible with the BCGroups strategy: the DisGB strategies achieve a latency similar to flooding while requiring significantly less inter-broker messages.
3. *MockFog: Automated Execution of Fog Application Experiments in the Cloud.* The main idea of MockFog is to emulate an infrastructure testbed in the cloud which can be manipulated based on a predefined orchestration schedule. This way, fog applications and fog systems can be deployed in the cloud while experiencing comparable performance and failure characteristics as in a real fog deployment. Moreover, it allows application engineers to test arbitrary failure scenarios and various infrastructure options at large scale, which is also not possible on small local testbeds.

This thesis is divided into four parts. In Part I, Foundations, we started with an introduction and necessary background information in Chapters 1 and 2. Chapter 2 also includes our generally applicable geo-context model. In Chapter 3, we discussed related work.

In Part II, Design & Implementation, we described the approaches related to the three main contributions, the corresponding system designs, and details on the respective proof-of-concept implementation prototypes. First, in Chapter 4, presented BCGroups, an inter-broker routing strategy for distributing IoT data within fog-based pub/sub systems. Second, in Chapter 5, we presented GeoBroker, a single node pub/sub broker system leveraging geo-context for IoT data distribution. Third, in Chapter 6, we presented DisGB, an extension of the GeoBroker system that leverages geo-context to additionally improve inter-broker routing. Fourth, in Chapter 7, we presented MockFog, an approach for the automated execution of fog application experiments in the cloud.

In Part III, Experiments, we evaluated the contributions in the same order that we used in Part II. For this, we described how we set up experiments and presented experiment results in Chapters 8 to 11.

In Part IV, Conclusion, we summarized the contributions in Chapter 12 and discuss our results and potential future research directions in Chapter 13.

Chapter 13

Discussion & Outlook

The experiments results indicate that our contributions significantly advance the state of the art for IoT data distribution with fog-based pub/sub systems, as well as the state of the art for testing and benchmarking of such systems and fog applications. In the following, we discuss general limitations of the contributions and potential future research directions. A detailed discussion on individual contributions was already presented in Sections 4.4, 5.5, 6.6 and 7.7.

The experiments and the simulation results show that the two DisGB strategies provide an event delivery latency comparable to event and subscription flooding. Furthermore, the simulation confirmed that DisGB requires less inter-broker messages than strategies from related work, including our own BCGroups strategy. This, however, is only the case if geo-context information is available. Without such information, the DisGB strategies require as many inter-broker messages as the flooding strategies. Therefore, when building a communication middleware for the IoT, we recommend using a hybrid approach: If geo-context information is available, RPs should be selected using either of the two DisGB strategies, in particular depending on whether clients update subscriptions or publish events more often. If not, we recommend using BCGroups to manage the latency and excess data dissemination tradeoff.

A promising topic for future work is to use the operation data of our pub/sub systems, e.g., message flows and client interactions, to extend approaches from the social networks field. Existing work, e.g., [11, 45, 52, 126, 194], often uses location traces collected from social media to identify correlation of users and locations, to derive recommendations, or to identify (emergency) events. In addition, they often have to identify which physical area is affected by a particular message/tweet/picture/blog post—extracting this information is, in many cases, not trivial. With GeoBroker and DisGB, this information is available out of the box with high precision; e.g., a publisher already defines an area of relevance with the event geofence. Furthermore, with the subscription geofence, subscribers can precisely describe the areas they are interested in, which is more accurate than estimating this based on locations and information extracted from, for example, tweets. Besides, our systems can also be used to spread targeted information and emergency warnings that were identified through data extracted from social networks [112, 187], e.g., a forest fire only affects people in a specific area. Therefore, a related emergency warning does only need to be distributed to people living in proximity to this area.

Regarding the emulation of benchmarking and testing infrastructure, MockFog does only work well when no specific local hardware[45] is required and when communication in the evaluated system is based on TCP/IP. MockFog also tends to work better for the emulation of larger edge machines such as a Raspberry Pi but faces limitations when smaller devices are involved as they cannot be emulated accurately. A pub/sub system usually operates within the core network or in the cloud, so machines are connected via wires or optic fiber, communicate via TCP/IP, and are sufficiently powerful. Thus, MockFog is a great fit for testing and evaluating our fog-based pub/sub system prototypes and inter-broker routing strategies. For other kinds of applications that, for example, comprise sensors which communicate via a LoRaWAN [169] such as TheThingsNetwork[46], MockFog's approach of emulating connections between devices does not work out of the box as these sensors expect to have access to a LoRa sender. As a solution, application developers could adapt their sensor software to use TCP/IP when no Lora sender is available, or use a library that already mocks LoRa-based communication in such cases.

Closing Remarks In this thesis, we proposed novel publish/subscribe system designs and inter-broker routing strategies that are specifically tailored for the fog and IoT data. In contrast to the current state of the art, our approaches use IoT-specific domain knowledge to optimize inter-broker routing and event delivery. This reduces communication latency between clients and brokers, as well as excess data dissemination. Furthermore, we proposed an approach for the automated execution of fog application experiments in the cloud which allowed us to experimentally evaluate our system designs and strategies without requiring access to a physical infrastructure testbed.

[45] E.g., some devices might have the use of a particular crypto chip deeply embedded in the application source code.
[46] `https://www.thethingsnetwork.org`

Appendix

Appendix A

List of Publications

Publications used in this Thesis

[79] Jonathan Hasenburg, Martin Grambow, David Bermbach. **MockFog 2.0: Automated Execution of Fog Application Experiments in the Cloud**. In: IEEE Transactions on Cloud Computing. IEEE 2021.

[73] Jonathan Hasenburg, David Bermbach. **DisGB: Using Geo-Context Information for Efficient Routing in Geo-Distributed Pub/Sub Systems**. In: Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing 2020 (UCC 2020). IEEE 2020.

[74] Jonathan Hasenburg, David Bermbach. **GeoBroker: A Pub/Sub Broker Considering Geo-Context Information**. In: Software Impacts. Elsevier 2020.

[77] Jonathan Hasenburg, David Bermbach. **Using Geo-context Information for Efficient Rendezvous-based Routing in Publish/Subscribe Systems**. In: Proceedings of the KuVS-Fachgespräch Fog Computing 2020 (FGFC 2020). TU Wien 2020.

[82] Jonathan Hasenburg, Florian Stanek, Florian Tschorsch, David Bermbach. **Managing Latency and Excess Data Dissemination in Fog-Based Publish/Subscribe Systems**. In: Proceedings of the Second IEEE International Conference on Fog Computing 2020 (ICFC 2020). IEEE 2020.

[75] Jonathan Hasenburg, David Bermbach. **GeoBroker: Leveraging Geo-Contexts for IoT Data Distribution**. In: Computer Communications. Elsevier 2020.

[76] Jonathan Hasenburg, David Bermbach. **Towards Geo-Context Aware IoT Data Distribution**. In: Service-Oriented Computing – ICSOC 2019 Workshops (ISYCC 2019). Springer 2019.

[81] Jonathan Hasenburg, Martin Grambow, Elias Grünewald, Sascha Huk, David Bermbach. **MockFog: Emulating Fog Computing Infrastructure in the Cloud**. In: Proceedings of the First IEEE International Conference on Fog Computing 2019 (ICFC 2019). IEEE 2019.

Other Publications

[133] Tobias Pfandzelter, Jonathan Hasenburg, David Bermbach. **Towards a Computing Platform for the LEO Edge**. In: Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys 2021). ACM 2021.

[132] Tobias Pfandzelter, Jonathan Hasenburg, David Bermbach. **From Zero to Fog: Efficient Engineering of Fog-based Internet of Things Applications**. In: Software: Practice and Experience. Wiley 2021.

[99] Ahmet-Serdar Karakaya, Jonathan Hasenburg, David Bermbach. **SimRa: Using Crowdsourcing to Identify Near Miss Hotspots in Bicycle Traffic**. In: Pervasive and Mobile Computing. Elsevier 2020.

[16] David Bermbach, Setareh Maghsudi, Jonathan Hasenburg, Tobias Pfandzelter. **Towards Auction-Based Function Placement in Serverless Fog Platforms**. In: Proceedings of the Second IEEE International Conference on Fog Computing 2020 (ICFC 2020). IEEE 2020.

[80] Jonathan Hasenburg, Martin Grambow, David Bermbach. **Towards a Replication Service for Data-Intensive Fog Applications**. In: Proceedings of the 35th ACM Symposium on Applied Computing, Posters Track (SAC 2020). ACM 2020.

[78] Jonathan Hasenburg, Martin Grambow, David Bermbach. **FBase: A Replication Service for Data-Intensive Fog Applications**. In: Technical Report MCC.2019.1. TU Berlin & ECDF, Mobile Cloud Computing Research Group. 2019.

[64] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, David Bermbach. **Is it Safe to Dockerize my Database Benchmark?**. In: Proceedings of the 34th ACM Symposium on Applied Computing, Posters Track (SAC 2019). ACM 2019.

[63] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, David Bermbach. **Dockerization Impacts in Database Performance Benchmarking**. In: Technical Report MCC.2018.1. TU Berlin & ECDF, Mobile Cloud Computing Research Group. 2018.

[84] Jonathan Hasenburg, Sebastian Werner, David Bermbach. **Supporting the Evaluation of Fog-based IoT Applications During the Design Phase**. In: Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things (M4IoT 2018). ACM 2018.

[83] Jonathan Hasenburg, Sebastian Werner, David Bermbach. **FogExplorer**. In: Proceedings of the 19th International Middleware Conference, Demos and Posters (Middleware 2018). ACM 2018.

[62] Martin Grambow, Jonathan Hasenburg, David Bermbach. **Public Video Surveillance: Using the Fog to Increase Privacy**. In: Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things (M4IoT 2018). ACM 2018.

Appendix B

List of Software Contributions

The following open source software prototypes have been developed as part of this thesis:

Broadcast Groups for Moquette – https://github.com/MoeweX/moquette**:** Extension of the popular MQTT broker implementation in Java to also support broadcast groups. Disclaimer: The implementation of this prototype was primarily carried out by our student Florian Stanek as part his Master's thesis [176].

Broadcast Group Simulation – https://github.com/MoeweX/broadcast-group-simulation**:** A simulation of the broadcast group group formation process.

(Distributed) GeoBroker – https://github.com/MoeweX/geobroker**:** A pub/sub broker system that leverages geo-context for data distribution between clients and for routing between distributed brokers.

DisGB-Simulation – https://github.com/MoeweX/DisGB-Simulation**:** A simulation of the DisGB inter-broker routing strategies to determine effects on latency and excess data. Also implements the broadcast group strategy and other strategies from related work for comparison.

MockFog 2.0 – https://github.com/MoeweX/MockFog2**:** A tool for the automated execution of fog application experiments in the cloud.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. First Edition. Cambridge University, 2008.

[2] Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Takayuki Kuroda. "Wide Area Network-scale Discovery and Data Dissemination in Data-centric Publish/Subscribe Systems". In: *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference*. ACM, 2015, pp. 1–2. DOI: 10.1145/2830894.2830900.

[3] Carlos Andrés Ramiro, Claudio Fiandrino, Alejandro Blanco Pizarro, Pablo Jiménez Mateo, Norbert Ludant, and Joerg Widmer. "OpenLEON: An End-to-End Emulator from the Edge Data Center to the Mobile Users". In: *Proceedings of the 12th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*. ACM, 2018, pp. 19–27. DOI: 10.1145/3267204.3267210.

[4] The Kubernetes Authors. *Kubernetes - Production-Grade Container Orchestration*. 2020. URL: https://kubernetes.io/ (visited on Dec. 2, 2020).

[5] Daniel Balasubramanian, Abhishek Dubey, William Otte, William Emfinger, Pranav Kumar, and Gabor Karsai. "A Rapid Testing Framework for a Mobile Cloud". In: *2014 25th IEEE International Symposium on Rapid System Prototyping*. IEEE, 2014, pp. 128–134. DOI: 10.1109/RSP.2014.6966903.

[6] Roberto Baldoni, Leonardo Querzoni, Sasu Tarkoma, and Antonino Virgillito. "Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey". In: *Middleware for Network Eccentric and Mobile Applications*. Ed. by Benoit Garbinato, Hugo Miranda, and Luis Rodrigues. Springer, 2009, pp. 219–244.

[7] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. *OASIS Standard MQTT Version 5.0*. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html (visited on Sept. 8, 2020).

[8] Ryohei Banno, Jingyu Sun, Masahiro Fujita, Susumu Takeuchi, and Kazuyuki Shudo. "Dissemination of Edge-Heavy Data on Heterogeneous MQTT Brokers". In: *2017 IEEE 6th International Conference on Cloud Networking*. IEEE, 2017, pp. 1–7. DOI: 10.1109/CloudNet.2017.8071523.

[9] Ryohei Banno, Susumu Takeuchi, Michiharu Takemoto, Tetsuo Kawano, Takashi Kambayashi, and Masato Matsuo. "Designing Overlay Networks for Handling Exhaust Data in a Distributed Topic-based Pub/Sub Architecture". In: *Journal of Information Processing* 23.2 (2015), pp. 105–116. DOI: 10.2197/ipsjjip.23.105.

[10] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology". In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 631–636. DOI: 10.1109/CCGRID.2010.72.

[11] Jie Bao, Yu Zheng, and Mohamed F. Mokbel. "Location-based and Preference-aware Recommendation using Sparse Geo-Social Networking Data". In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. ACM, 2012, pp. 199–208. DOI: 10.1145/2424321.2424348.

[12] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. "StreamHub: A Massively Parallel Architecture for High-performance Content-based Publish/Subscribe". In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. ACM, 2013, pp. 63–74. DOI: 10.1145/2488222.2488260.

[13] Ilja Behnke, Lauritz Thamsen, and Odej Kao. "Héctor: A Framework for Testing IoT Applications Across Heterogeneous Edge and Cloud Testbeds". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. ACM, 2019, pp. 15–20. DOI: 10.1145/3368235.3368832.

[14] Jossekin Beilharz, Philipp Wiesner, Arne Boockmeyer, Florian Brokhausen, Ilja Behnke, Robert Schmid, Lukas Pirl, and Lauritz Thamsen. "Towards a Staging Environment for the Internet of Things". In: *2021 IEEE International Conference on Pervasive Computing and Communications Workshop*. IEEE, 2021. URL: http://arxiv.org/abs/2101.10697.

[15] Paolo Bellavista, Antonio Corradi, and Andrea Reale. "Quality of Service in Wide Scale Publish-Subscribe Systems". In: *IEEE Communications Surveys & Tutorials* 16.3 (2014), pp. 1591–1616. DOI: 10.1109/SURV.2014.031914.00192.

[16] David Bermbach, Setareh Maghsudi, Jonathan Hasenburg, and Tobias Pfandzelter. "Towards Auction-Based Function Placement in Serverless Fog Platforms". In: *2020 IEEE International Conference on Fog Computing*. IEEE, 2020, pp. 25–31. DOI: 10.1109/ICFC49376.2020.00012.

[17] David Bermbach, Frank Pallas, David García Pérez, Pierluigi Plebani, Maya Anderson, Ronen Kat, and Stefan Tai. "A Research Perspective on Fog Computing". In: *2nd Workshop on IoT Systems Provisioning & Management for Context-Aware Smart Cities*. Springer, 2018, pp. 198–210. DOI: 10.1007/978-3-319-91764-1_16.

[18] David Bermbach, Erik Wittern, and Stefan Tai. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. First Edition. Springer, 2017.

[19] David Bermbach, Liang Zhao, and Sherif Sakr. "Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services". In: *Technology Conference on Performance Characterization and Benchmarking*. Springer, 2014, pp. 32–47. DOI: 10.1007/978-3-319-04936-6_3.

[20] Bert Hubert. *tc - Traffic Control*. 2021. URL: https://man7.org/linux/man-pages/man8/tc.8.html (visited on Feb. 26, 2021).

[21] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. “Fog Computing: A Platform for Internet of Things and Analytics”. In: *Big Data and Internet of Things: A Roadmap for Smart Environments*. Ed. by Nik Bessis and Ciprian Dobre. Springer, 2014, pp. 169–186. DOI: 10.1007/978-3-319-05029-4_7.

[22] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. “Fog Computing and its Role in the Internet of Things”. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. ACM, 2012, pp. 13–16. DOI: 10.1145/2342509.2342513.

[23] Boris Beizer. *Software Testing Techniques*. 2nd ed. Van Nostrand Reinhold, 1990.

[24] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. “Integration of Cloud Computing and Internet of Things: A survey”. In: *Future Generation Computer Systems* 56 (2016), pp. 684–700. DOI: 10.1016/j.future.2015.09.021.

[25] Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, and Francesco Zanichelli. “A Simulation Platform for Large-Scale Internet of Things Scenarios in Urban Environments”. In: *Proceedings of the The First International Conference on IoT in Urban Space*. ICST, 2014, pp. 50–55. DOI: 10.4108/icst.urb-iot.2014.257268.

[26] Antonio Brogi, Stefano Forti, and Marco Gaglianese. “Measuring the Fog, Gently”. In: *Service-Oriented Computing*. Springer, 2019, pp. 523–538. DOI: 10.1007/978-3-030-33702-5_40.

[27] Robert Bryce, Thomas Shaw, and Gautam Srivastava. “MQTT-G: A Publish/Subscribe Protocol with Geolocation”. In: *2018 IEEE 41st International Conference on Telecommunications and Signal Processing*. IEEE, 2018, pp. 1–4. DOI: 10.1109/TSP.2018.8441479.

[28] Fengyun Cao and Jaswinder Pal Singh. “MEDYM: Match-Early with Dynamic Multicast for Content-Based Publish-Subscribe Networks”. In: *Middleware 2005*. Springer, 2005, pp. 292–313. DOI: 10.1007/11587552_15.

[29] Antonio Carzaniga. “Design and Evaluation of a Wide-Area Event Notification Service”. In: *ACM Transactions on Computer Systems* 19.3 (2001), pp. 332–383. DOI: 10.1145/380749.380767.

[30] Henri Casanova, Arnaud Legrand, and Martin Quinson. “SimGrid: a Generic Framework for Large-Scale Distributed Experiments”. In: *Tenth International Conference on Computer Modeling and Simulation*. IEEE, 2008, pp. 126–131. DOI: 10.1109/UKSIM.2008.28.

[31] Konstantinos Chaitas. “Design and Implementation of a Scalable, Distributed, Location-Based Pub/Sub System”. MA thesis. Technische Universität Berlin, 2020.

[32] Bertil Chapuis and Benoit Garbinato. “Scaling and Load Testing Location-Based Publish and Subscribe”. In: *IEEE 37th International Conference on Distributed Computing Systems*. IEEE, 2017, pp. 2543–2546. DOI: 10.1109/ICDCS.2017.234.

[33] Bertil Chapuis, Benoit Garbinato, and Lucas Mourot. “A Horizontally Scalable and Reliable Architecture for Location-Based Publish-Subscribe”. In: *IEEE 36th Symposium on Reliable Distributed Systems*. IEEE, 2017, pp. 74–83. DOI: 10.1109/SRDS.2017.16.

[34] Xiaoyan Chen, Ying Chen, and Fangyan Rao. "An Efficient Spatial Publish/Subscribe System for Intelligent Location-based Services". In: *2nd International Workshop on Distributed Event-based Systems*. ACM, 2003, pp. 1–6. DOI: 10.1145/966618.966625.

[35] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. "SpiderCast: A Scalable Interest-aware Overlay for Topic-based Pub/Sub Communication". In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. ACM, 2007, pp. 14–25. DOI: 10.1145/1266894.1266899.

[36] Chi-Yin Chow, Jie Bao, and Mohamed F. Mokbel. "Towards Location-based Social Networking Services". In: *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Location Based Social Networks*. ACM, 2010, pp. 31–38. DOI: 10.1145/1867699.1867706.

[37] Cisco Networking Academy. *Cisco Packet Tracer - Networkin Simulation Tool*. 2021. URL: https://www.netacad.com/courses/packet-tracer (visited on Feb. 26, 2021).

[38] Brian Frank Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154. DOI: 10.1145/1807128.1807152.

[39] Antonio Coutinho, Fabıola Greve, Cassio Prazeres, and Joao Cardoso. "Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing". In: *2018 IEEE International Conference on Communications*. IEEE, 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8423003.

[40] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. "Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems". In: *20th International Conference on Software Engineering*. IEEE, 1998, pp. 261–270. DOI: 10.1109/ICSE.1998.671135.

[41] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. "The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS". In: *IEEE Transactions on Software Engineering* 27.9 (2001), pp. 827–850. DOI: 10.1109/32.950318.

[42] Gianpaolo Cugola and Jose Enrique Munoz de Cote. "On Introducing Location Awareness in Publish-Subscribe Middleware". In: *25th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, 2005, pp. 377–382. DOI: 10.1109/ICDCSW.2005.101.

[43] Docker. *Docker - Empowering App Development for Developers*. 2020. URL: https://docker.com (visited on Sept. 9, 2020).

[44] Docker. *Docker Update - Docker Documentation*. 2021. URL: https://docs.docker.com/engine/reference/commandline/update/ (visited on Feb. 26, 2021).

[45] Yerach Doytsher, Ben Galon, and Yaron Kanza. "Querying Geo-Social Data by Bridging Spatial Networks and Social Networks". In: *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Location Based Social Networks - LBSN '10*. ACM, 2010, pp. 39–46. DOI: 10.1145/1867699.1867707.

[46] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud C. Narendra, and Bo Hu. "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015, pp. 621–628. DOI: 10.1109/CLOUD.2015.88.

[47] Eclipse Foundation. *Eclipse Mosquitto - An Open Source MQTT Broker*. 2021. URL: https://mosquitto.org/ (visited on Feb. 26, 2021).

[48] Eclipse Foundation. *Eclipse Paho MQTT Client*. 2021. URL: https://www.eclipse.org/paho/ (visited on Feb. 26, 2021).

[49] Scott Eisele, Geoffrey Pettet, Abhishek Dubey, and Gabor Karsai. "Towards an Architecture for Evaluating and Analyzing Decentralized Fog Applications". In: *2017 IEEE Fog World Congress*. IEEE, 2017, pp. 1–6. DOI: 10.1109/FWC.2017.8368531.

[50] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The many Faces of Publish/Subscribe". In: *ACM Computing Surveys* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078.

[51] USA Federal Communications Commission. *Wireless Emergency Alerts*. 2020. URL: https://www.fcc.gov/consumers/guides/wireless-emergency-alerts-wea (visited on Sept. 18, 2020).

[52] Laura Ferrari, Alberto Rosi, Marco Mamei, and Franco Zambonelli. "Extracting Urban Patterns from Location-based Social Networks". In: *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Location-Based Social Networks - LBSN '11*. ACM, 2011, pp. 9–16. DOI: 10.1145/2063212.2063226.

[53] Ludger Fiege, Felix C. Gartner, Oliver Kasten, and Andreas Zeidler. "Supporting Mobility in Content-Based Publish/Subscribe Middleware". In: *Middleware 2003*. Vol. 2672. Springer, 2003, pp. 103–122. DOI: 10.1007/3-540-44892-6_6.

[54] Foamspace Corporation. *FOAM Whitepaper*. 2018. URL: https://www.foam.space/publicAssets/FOAM_Whitepaper.pdf (visited on Dec. 16, 2020).

[55] Ramon R. Fontes, Samira Afzal, Samuel H. B. Brito, Mateus A. S. Santos, and Christian Esteve Rothenberg. "Mininet-WiFi: Emulating Software-defined Wireless Networks". In: *2015 11th International Conference on Network and Service Management*. IEEE, 2015, pp. 384–389. DOI: 10.1109/CNSM.2015.7367387.

[56] Davide Frey and Gruia-Catalin Roman. "Context-Aware Publish Subscribe in Mobile Ad Hoc Networks". In: *Proceedings of the International Conference on Coordination Languages and Models*. Ed. by Amy L. Murphy and Jan Vitek. Vol. 4467. Springer, 2007, pp. 37–55. DOI: 10.1007/978-3-540-72794-1_3.

[57] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys & Tutorials* 17.4 (2015), pp. 2347–2376. DOI: 10.1109/COMST.2015.2444095.

[58] Julien Gascon-Samson, Franz-Philippe Garcia, Bettina Kemme, and Jorg Kienzle. "Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 486–496. DOI: 10.1109/ICDCS.2015.56.

[59] Julien Gascon-Samson, Jörg Kienzle, and Bettina Kemme. "MultiPub: Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe". In: *IEEE 37th International Conference on Distributed Computing Systems*. IEEE, 2017, pp. 2075–2082. DOI: 10.1109/ICDCS.2017.203.

[60] GitHub Contributors. *Moquette MQTT Broker*. 2021. URL: https://moquette-io.github.io/moquette/ (visited on Feb. 26, 2021).

[61] Google. *Compute Engine - Virtual Machines (VMs) | Google Cloud*. 2020. URL: https://cloud.google.com/compute (visited on Dec. 11, 2020).

[62] Martin Grambow, Jonathan Hasenburg, and David Bermbach. "Public Video Surveillance: Using the Fog to Increase Privacy". In: *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things*. ACM, 2018, pp. 11–14. DOI: 10.1145/3286719.3286722.

[63] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, and David Bermbach. "Dockerization Impacts in Database Performance Benchmarking". In: *Technical Report MCC.2018.1*. TU Berlin & ECDF, Mobile Cloud Computing Research Group, 2018, pp. 2–9. URL: https://arxiv.org/abs/arXiv:1812.04362v1.

[64] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, and David Bermbach. "Is it Safe to Dockerize my Database Benchmark?" In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2019, pp. 341–344. DOI: 10.1145/3297280.3297545.

[65] Matti Grotheer. *Experiment-Driven Assessment of Spatio-Textual Index Data Structures*. BA Thesis. Technische Universität Berlin. 2019.

[66] Long Guo, Lu Chen, Dongxiang Zhang, Guoliang Li, Kian-Lee Tan, and Zhifeng Bao. "Elaps: An Efficient Location-Aware Pub/Sub System". In: *IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1504–1507. DOI: 10.1109/ICDE.2015.7113412.

[67] Long Guo, Dongxiang Zhang, Guoliang Li, Kian-Lee Tan, and Zhifeng Bao. "Location-Aware Pub/Sub System: When Continuous Moving Queries Meet Dynamic Event Streams". In: *ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 843–857. DOI: 10.1145/2723372.2746481.

[68] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. "Meghdoot: Content-Based Publish/Subscribe over P2P Networks". In: *Middleware 2004*. Ed. by Hans-Arno Jacobsen. Springer, 2004, pp. 254–273. DOI: 10.1007/978-3-540-30229-2_14.

[69] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. "iFogSim: A Toolkit for Modeling and Simulation of Resource Management Techniques in the Internet of Things, Edge and Fog Computing Environments". In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296. DOI: 10.1002/spe.2509.

[70] Marios Hadjieleftheriou, Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Encyclopedia of GIS*. Ed. by Shashi Shekhar, Hui Xiong, and Xun Zhou. Springer, 2017, pp. 1805–1817. DOI: 10.1007/978-3-319-17885-1_1151.

[71] Toshihiro Hanawa, Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, and Mitsuhisa Sato. "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems". In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 428–433. DOI: 10.1109/ICSTW.2010.59.

[72] Daniel Happ, Niels Karowski, Thomas Menzel, Vlado Handziski, and Adam Wolisz. "Meeting IoT Platform Requirements with Open Pub/Sub Solutions". In: *Springer Annals of Telecommunications* 72.1 (2017), pp. 41–52. DOI: 10.1007/s12243-016-0537-4.

[73] Jonathan Hasenburg and David Bermbach. "DisGB: Using Geo-Context Information for Efficient Routing in Geo-Distributed Pub/Sub Systems". In: *2020 IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE, 2020, pp. 67–78. DOI: 10.1109/UCC48980.2020.00026.

[74] Jonathan Hasenburg and David Bermbach. "GeoBroker: A Pub/Sub Broker Considering Geo-Context Information". In: *Software Impacts* 6 (2020), pp. 100029–100031. DOI: 10.1016/j.simpa.2020.100029.

[75] Jonathan Hasenburg and David Bermbach. "GeoBroker: Leveraging Geo-Contexts for IoT Data Distribution". In: *Computer Communications* 151 (2020), pp. 473–484. DOI: 10.1016/j.comcom.2020.01.015.

[76] Jonathan Hasenburg and David Bermbach. "Towards Geo-Context Aware IoT Data Distribution". In: *Service-Oriented Computing - ICSOC 2019 Workshops*. Springer, 2019, pp. 111–121. DOI: 10.1007/978-3-030-45989-5_9.

[77] Jonathan Hasenburg and David Bermbach. "Using Geo-Context Information for Efficient Rendezvous-based Routing in Publish/Subscribe Systems". In: *KuVS-Fachgespräch Fog Computing 2020*. In collab. with Zoltán Ádám Mann and Schulte, Stefan. TU Wien, 2020, pp. 4–7. DOI: 10.34726/kuvs2020.

[78] Jonathan Hasenburg, Martin Grambow, and David Bermbach. "FBase: A Replication Service for Data-Intensive Fog Applications". In: *Technical Report MCC.2019.1*. TU Berlin & ECDF, Mobile Cloud Computing Research Group, 2019, pp. 2–11. URL: https://arxiv.org/abs/arXiv:1912.03107v2.

[79] Jonathan Hasenburg, Martin Grambow, and David Bermbach. "MockFog 2.0: Automated Execution of Fog Application Experiments in the Cloud". In: *IEEE Transactions on Cloud Computing* Early Access (2021). DOI: 10.1109/TCC.2021.3074988.

[80] Jonathan Hasenburg, Martin Grambow, and David Bermbach. "Towards a Replication Service for Data-Intensive Fog Applications". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. ACM, 2020, pp. 267–270. DOI: 10.1145/3341105.3374060.

[81] Jonathan Hasenburg, Martin Grambow, Elias Grunewald, Sascha Huk, and David Bermbach. "MockFog: Emulating Fog Computing Infrastructure in the Cloud". In: *2019 IEEE International Conference on Fog Computing*. IEEE, 2019, pp. 144–152. DOI: 10.1109/ICFC.2019.00026.

[82] Jonathan Hasenburg, Florian Stanek, Florian Tschorsch, and David Bermbach. “Managing Latency and Excess Data Dissemination in Fog-Based Publish/Subscribe Systems”. In: *2020 IEEE International Conference on Fog Computing*. IEEE, 2020, pp. 9–16. DOI: 10.1109/ICFC49376.2020.00010.

[83] Jonathan Hasenburg, Sebastian Werner, and David Bermbach. “FogExplorer”. In: *Proceedings of the 19th International Middleware Conference (Posters)*. Middleware ’18. ACM, 2018, pp. 1–2. DOI: 10.1145/3284014.3284015.

[84] Jonathan Hasenburg, Sebastian Werner, and David Bermbach. “Supporting the Evaluation of Fog-based IoT Applications During the Design Phase”. In: *Proceedings of the 5th Workshop on Middlware and Applications for the Internet of Things*. ACM, 2018, pp. 1–6. DOI: 10.1145/3286719.3286720.

[85] Raoufeh Hashemian, Niklas Carlsson, Diwakar Krishnamurthy, and Martin Arlitt. “WoTbench: A Benchmarking Framework for the Web of Things”. In: *Proceedings of the 9th International Conference on the Internet of Things*. ACM, 2019, pp. 1–4. DOI: 10.1145/3365871.3365897.

[86] Raoufehsadat Hashemian, Niklas Carlsson, Diwakar Krishnamurthy, and Martin Arlitt. “Contention Aware Web of Things Emulation Testbed”. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2020, pp. 246–256. DOI: 10.1145/3358960.3379140.

[87] Stefan Herle, Ralf Becker, and Jörg Blankenbach. “Bridging GeoMQTT and REST”. Proceedings of the Geospatial Sensor Webs Conference. 2016.

[88] Stefan Herle and Jörg Blankenbach. “Enhancing the OGC WPS Interface with GeoPipes Support for Real-Time Geoprocessing”. In: *International Journal of Digital Earth* 11.1 (2018), pp. 48–63. DOI: 10.1080/17538947.2017.1319976.

[89] Pieter Hintjens. *ZeroMQ: Messaging for many Applications*. First Edition. O’Reilly Media Inc., 2013.

[90] Bradley Huffaker, Marina Fomenkov, Daniel J. Plummer, and David Moore. “Distance Metrics in the Internet”. IEEE International Telecommunications Symposium. 2002.

[91] Alefiya Hussain, Prateek Jaipuria, Geoff Lawler, Stephen Schwab, and Terry Benzel. “Toward Orchestration of Complex Networking Experiments”. In: *13th USENIX Workshop on Cyber Security Experimentation and Test*. USENIX, 2020.

[92] Transforma Insights. *Global IoT Market by 2030*. 2020. URL: https://transformainsights.com/news/iot-market-24-billion-usd15-trillion-revenue-2030 (visited on Sept. 28, 2020).

[93] Raj Jain and Imrich Chlamtac. “The P2 Algorithm for Dynamic Calculation of Quantiles and Histograms without Storing Observations”. In: *Communications of the ACM* 28.10 (1985), pp. 1076–1085. DOI: 10.1145/4372.4378.

[94] Jetbrains. *Kotlin Multiplatform / Multi-format Reflectionless Serialization*. 2020. URL: https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serialization-guide.md (visited on Oct. 21, 2020).

[95] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. "Characterizing Data Analysis Workloads in Data Centers". In: *2013 IEEE International Symposium on Workload Characterization*. IEEE, 2013, pp. 66–76. DOI: 10.1109/IISWC.2013.6704671.

[96] Paul C Jorgensen. *Software Testing*. Fourth Edition. CRC, 2014.

[97] Algimantas Kajackas and Rytis Rainys. "Internet Infrastructure Topology Assessment". In: *Electronics and Electrical Engineering* 7.103 (2010), pp. 91–94.

[98] Vasileios Karagiannis and Stefan Schulte. "Comparison of Alternative Architectures in Fog Computing". In: *2020 IEEE 4th International Conference on Fog and Edge Computing*. IEEE, 2020, pp. 19–28. DOI: 10.1109/ICFEC50348.2020.00010.

[99] Ahmet-Serdar Karakaya, Jonathan Hasenburg, and David Bermbach. "SimRa: Using Crowdsourcing to Identify Near Miss Hotspots in Bicycle Traffic". In: *Pervasive and Mobile Computing* 67 (2020), pp. 101197–101208. DOI: 10.1016/j.pmcj.2020.101197.

[100] David Kargerl, Eric Lehmanl, Tom Leightonl', Rina Panigrahyl, and Daniel Lewinl. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *29th ACM Symposium on Theory of Computing*. ACM, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[101] Zakaria Kasmi, Naouar Guerchali, Abdelmoumen Norrdine, and Jochen H. Schiller. "Algorithms and Position Optimization for a Decentralized Localization Platform Based on Resource-Constrained Devices". In: *IEEE Transactions on Mobile Computing* 18.8 (2019), pp. 1731–1744. DOI: 10.1109/TMC.2018.2868930.

[102] Ryo Kawaguchi and Masaki Bandai. "A Distributed MQTT Broker System for Location-based IoT Applications". In: *2019 IEEE International Conference on Consumer Electronics*. IEEE, 2019, pp. 1–4. DOI: 10.1109/ICCE.2019.8662069.

[103] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. "Reliable and Highly Available Distributed Publish/Subscribe Service". In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 41–50. DOI: 10.1109/SRDS.2009.32.

[104] Shweta Khare, Hongyang Sun, Julien Gascon-Samson, Kaiwen Zhang, Aniruddha Gokhale, Yogesh Barve, Anirban Bhattacharjee, and Xenofon Koutsoukos. "Linearize, Predict and Place: Minimizing the Makespan for Edge-based Stream Processing of Directed Acyclic Graphs". In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. ACM, 2019, pp. 1–14. DOI: 10.1145/3318216.3363315.

[105] Abdelmajid Khelil and David Soldani. "On the Suitability of Device-to-Device Communications for Road Traffic Safety". In: *IEEE World Forum on Internet of Things*. IEEE, 2014, pp. 224–229. DOI: 10.1109/WF-IoT.2014.6803163.

[106] Srdjan Kr and Francois Carrez. "Designing IoT Architecture(s): A European Perspective". In: *2014 IEEE World Forum on Internet of Things*. IEEE, 2014, pp. 79–84. DOI: 10.1109/WF-IoT.2014.6803124.

[107] Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach". In: *2003 VLDB Conference*. Elsevier, 2003, pp. 608–619. DOI: 10.1016/B978-012722442-8/50060-4.

[108] Isaac Lera, Carlos Guerrero, and Carlos Juiz. “YAFS: A Simulator for IoT Scenarios in Fog Computing”. In: *IEEE Access* 7 (2019), pp. 91745–91758. DOI: 10.1109/ACCESS.2019.2927895.

[109] Guoliang Li, Yang Wang, Ting Wang, and Jianhua Feng. “Location-aware Publish/Subscribe”. In: *19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 802–810. DOI: 10.1145/2487575.2487617.

[110] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. “Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Cloud-Scale Infrastructure”. In: *17th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2020, pp. 389–402.

[111] Jason Liu. “Parallel Discrete-Event Simulation”. In: *Wiley Encyclopedia of Operations Research and Management Science*. American Cancer Society, 2011. URL: https://doi.org/10.1002/9780470400531.eorms0639.

[112] Bertrand De Longueville, Robin S. Smith, and Gianluca Luraschi. “"OMG, from here, I can see the Flames!": A Use Case of Mining Location-based Social Networks to Acquire Spatio-Temporal Data on Forest Fires”. In: *Proceedings of the 2009 International Workshop on Location based Social Networks*. ACM, 2009, pp. 73–80. DOI: 10.1145/1629890.1629907.

[113] Vilen Looga, Zhonghong Ou, Yang Deng, and Antti Yla-Jaaski. “MAMMOTH: A Massive-Scale Emulation Platform for Internet of Things”. In: *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*. IEEE, 2012, pp. 1235–1239. DOI: 10.1109/CCIS.2012.6664581.

[114] Roger Lott. *Geographic Information-Well-Known Text Representation of Coordinate Reference Systems*. 2015. URL: http://docs.opengeospatial.org/is/12-063r5/12-063r5.html (visited on Dec. 7, 2020).

[115] Chunjie Luo, Jianfeng Zhan, Zhen Jia, Lei Wang, Gang Lu, Lixin Zhang, Cheng-Zhong Xu, and Ninghui Sun. “CloudRank-D: Benchmarking and Ranking Cloud Computing Systems for Data Processing Applications”. In: *Frontiers of Computer Science* 6.4 (2012), pp. 347–362. DOI: 10.1007/s11704-012-2118-7.

[116] Harsha V. Madhyastha. *iPlane: An Information Plane for Distributed Services*. 2020. URL: https://web.eecs.umich.edu/~harshavm/iplane/ (visited on Oct. 7, 2020).

[117] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. “Fog Computing: A Taxonomy, Survey and Future Directions”. In: *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*. Ed. by Beniamino Di Martino, Kuan-Ching Li, Laurence T. Yang, and Antonio Esposito. Springer, 2018, pp. 103–130. URL: https://doi.org/10.1007/978-981-10-5861-5_5.

[118] Jose Legatheaux Martins and Sergio Duarte. “Routing Algorithms for Content-based Publish/-Subscribe Systems”. In: *IEEE Communications Surveys & Tutorials* 12.1 (2010), pp. 39–58. DOI: 10.1109/SURV.2010.020110.00065.

[119] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. "EmuFog: Extensible and Scalable Emulation of Large-Scale Fog Computing Infrastructures". In: *2017 IEEE Fog World Congress*. IEEE, 2017, pp. 1–6. DOI: https://doi.org/10.1109/FWC.2017.8368525.

[120] Jonathan McChesney, Nan Wang, Ashish Tanwer, Eyal de Lara, and Blesson Varghese. "DeFog: Fog Computing Benchmarks". In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. ACM, 2019, pp. 47–58. DOI: 10.1145/3318216.3363299.

[121] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology Special Publication 800-145. 2011.

[122] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud*. 1st ed. O'Reilly, 2016.

[123] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. "A Serverless Real-Time Data Analytics Platform for Edge Computing". In: *IEEE Internet Computing* 21.4 (2017), pp. 64–71. DOI: 10.1109/MIC.2017.2911430.

[124] Hang Nguyen, Md Yusuf Sarwar Uddin, and Nalini Venkatasubramanian. "Multistage Adaptive Load Balancing for Big Active Data Publish Subscribe Systems". In: *13th ACM International Conference on Distributed and Event-based Systems*. ACM, 2019, pp. 43–54. DOI: 10.1145/3328905.3329508.

[125] Abdelmoumen Norrdine, Zakaria Kasmi, Kashan Ahmed, Christoph Motzko, and Jochen Schiller. "MQTT-Based Surveillance System of IoT Using UWB Real Time Location System". In: *2020 International Conferences on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data and IEEE Congress on Cybermatics*. IEEE, 2020, pp. 216–221. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics50389.2020.00050.

[126] Anastasios Noulas, Salvatore Scellato, Cecilia Mascolo, and Massimiliano Pontil. "Exploiting Semantic Annotations for Clustering Geographic Areas and Users in Location-Based Social Networks". Fifth International AAAI Conference on Weblogs and Social Media. 2011.

[127] Rogerio Leao Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. "Using Mininet for Emulation and Prototyping Software-defined Networks". In: *2014 IEEE Colombian Conference on Communications and Computing*. IEEE, 2014, pp. 1–6. DOI: 10.1109/ColComCon.2014.6860404.

[128] Helge Parzyjegla, Daniel Graff, Arnd Schröter, Jan Richling, and Gero Mühl. "Design and Implementation of the Rebeca Publish/Subscribe Middleware". In: *From Active Data Management to Event-Based Systems and More* 6462 (2010), pp. 124–140. DOI: 10.1007/978-3-642-17226-7_8.

[129] Manuel Peuster, Johannes Kampmeyer, and Holger Karl. "Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains". In: *2018 4th IEEE Conference on Network Softwarization and Workshops*. IEEE, 2018, pp. 335–337. DOI: 10.1109/NETSOFT.2018.8459905.

[130] Manuel Peuster, Holger Karl, and Steven van Rossem. “MeDICINE: Rapid Prototyping of Production-ready Network Services in multi-PoP Environments”. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks*. IEEE, 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490.

[131] Tobias Pfandzelter and David Bermbach. “IoT Data Processing in the Fog: Functions, Streams, or Batch Processing?” In: *2019 IEEE International Conference on Fog Computing*. IEEE, 2019, pp. 201–206. DOI: 10.1109/ICFC.2019.00033.

[132] Tobias Pfandzelter, Jonathan Hasenburg, and David Bermbach. “From Zero to Fog: Efficient Engineering of Fog-based Internet of Things Applications”. In: *Software: Practice and Experience* (2021), spe.3003. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.3003.

[133] Tobias Pfandzelter, Jonathan Hasenburg, and David Bermbach. “Towards a Computing Platform for the LEO Edge”. In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. ACM, 2021, pp. 43–48. DOI: 10.1145/3434770.3459736.

[134] Peter Pietzuch and Jean Bacon. “Hermes: A Distributed Event-based Middleware Architecture”. In: *22nd International Conference on Distributed Computing Systems Workshops*. IEEE, 2002, pp. 611–618. DOI: 10.1109/ICDCSW.2002.1030837.

[135] Jon Postel. *Internet Control Message Protocol*. RFC 792. 1981. DOI: 10.17487/RFC0792.

[136] Alina Quereilhac, Mathieu Lacage, Claudio Freire, Thierry Turletti, and Walid Dabbous. “NEPI: An Integration Framework for Network Experimentation”. In: *19th International Conference on Software, Telecommunications and Computer Networks*. IEEE, 2011, pp. 1–5.

[137] Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H. Payberah, and Seif Haridi. “Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks”. In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 746–757. DOI: 10.1109/IPDPS.2011.75.

[138] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. “OMF: A Control and Management Framework for Networking Testbeds”. In: *ACM SIGOPS Operating Systems Review* 43.4 (2010), pp. 54–59. DOI: 10.1145/1713254.1713267.

[139] Brian Ramprasad, Marios Fokaefs, Joydeep Mukherjee, and Marin Litoiu. “EMU-IoT - A Virtual Internet of Things Lab”. In: *2019 IEEE International Conference on Autonomic Computing*. IEEE, 2019, pp. 73–83. DOI: 10.1109/ICAC.2019.00019.

[140] Brian Ramprasad, Joydeep Mukherjee, and Marin Litoiu. “A Smart Testing Framework for IoT Applications”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*. IEEE, 2018, pp. 252–257. DOI: 10.1109/UCC-Companion.2018.00064.

[141] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Schahram Dustdar, and Philipp Raith. “Synthesizing Plausible Infrastructure Configurations for Evaluating Edge Computing Systems”. In: *3rd USENIX Workshop on Hot Topics in Edge Computing*. USENIX, 2020.

[142] Thomas Rausch, Stefan Nastic, and Schahram Dustdar. “EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications”. In: *2018 IEEE International Conference on Cloud Engineering*. IEEE, 2018, pp. 191–197. DOI: 10.1109/IC2E.2018.00043.

[143] Fabrice Reclus and Kristen Drouard. "Geofencing for Fleet & Freight Management". In: *9th International Conference on Intelligent Transport Systems Telecommunications*. IEEE, 2009, pp. 353–356. DOI: 10.1109/ITST.2009.5399328.

[144] Red Hat. *Ansible - Drive Automation across Open Hybrid Cloud Deployments*. 2021. URL: https://ansible.com (visited on Feb. 26, 2021).

[145] Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. "Internet of Things Patterns". In: *Proceedings of the 21st European Conference on Pattern Languages of Programs*. ACM, 2016, pp. 1–21. DOI: 10.1145/3011784.3011789.

[146] Ju Ren, Hui Guo, Chugui Xu, and Yaoxue Zhang. "Serving at the Edge: A Scalable IoT Architecture Based on Transparent Computing". In: *IEEE Network* 31.5 (2017), pp. 96–105. DOI: 10.1109/MNET.2017.1700030.

[147] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. "Scribe: The Design of a Large-Scale Event Notification Infrastructure". In: *International Workshop on Networked Group Communication*. Springer, 2001, pp. 30–43. DOI: 10.1007/3-540-45546-9_3.

[148] Mohd Ezanee Rusli, Mohammad Ali, Norziana Jamil, and Marina Md Din. "An Improved Indoor Positioning Algorithm Based on RSSI-Trilateration Technique for Internet of Things (IOT)". In: *2016 International Conference on Computer and Communication Engineering*. IEEE, 2016, pp. 72–77. DOI: 10.1109/ICCCE.2016.28.

[149] Maria Salama, Yehia Elkhatib, and Gordon Blair. "IoTNetSim: A Modelling and Simulation Platform for End-to-End IoT Services and Networking". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. ACM, 2019, pp. 251–261. DOI: 10.1145/3344341.3368820.

[150] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R. Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, and Dennis Pfisterer. "SmartSantander: IoT Experimentation over a Smart City Testbed". In: *Computer Networks* 61 (2014), pp. 217–238. DOI: 10.1016/j.bjp.2013.12.020.

[151] Lidiane Santos, Jorge Pereira, Eduardo Silva, Thais Batista, Everton Cavalcante, and Jair Leite. "Identifying Requirements for Architectural Modeling in Internet of Things Applications". In: *2019 IEEE International Conference on Software Architecture Companion*. IEEE, 2019, pp. 19–26. DOI: 10.1109/ICSA-C.2019.00011.

[152] Lidiane Santos, Eduardo Silva, Thais Batista, Everton Cavalcante, Jair Leite, and Flavio Oquendo. "An Architectural Style for Internet of Things Systems". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. ACM, 2020, pp. 1488–1497. DOI: 10.1145/3341105.3374030.

[153] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. "Design and Evaluation of a Virtual Experimental Environment for Distributed Systems". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 172–179. DOI: 10.1109/PDP.2013.32.

[154] Sasu Tarkoma. *Publish/Subscribe Systems - Design and Principles*. First Edition. Wiley Series in Communications Networking & Distributed Systems. Wiley, 2012.

[155] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. "The Case for VM-based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.64.

[156] Mahadev Satyanarayanan, Grace Lewis, Edwin Morris, Soumya Simanta, Jeff Boleng, and Kiryong Ha. "The Role of Cloudlets in Hostile Environments". In: *IEEE Pervasive Computing* 12.4 (2013), pp. 40–49. ISSN: 1536-1268. DOI: 10.1109/MPRV.2013.77.

[157] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C. Gall. "Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies". In: *Proceedings of the 17th International Middleware Conference*. ACM, 2016, pp. 1–14. DOI: 10.1145/2988336.2988348.

[158] Sejin Shun, Sangjin Shin, Seungmin Seo, Sungkwang Eom, Jooik Jung, and Kyong-Ho Lee. "A Pub/Sub-based Fog Computing Architecture for Internet-of-Vehicles". In: *2016 IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2016, pp. 90–93. DOI: 10.1109/CloudCom.2016.0029.

[159] Amazon Web Services. *Amazon CloudWatch - Observability of your AWS Resources and Applications on AWS and On-Premises*. 2021. URL: https://aws.amazon.com/cloudwatch (visited on Feb. 26, 2021).

[160] Amazon Web Services. *Amazon EC2 - Secure and Resizable Compute Capacity to Support Virtually Any Workload*. 2020. URL: https://aws.amazon.com/ec2/ (visited on Oct. 21, 2020).

[161] Amazon Web Services. *AWS Greengrass - Bring Local Compute, Messaging, Data Management, Sync, and ML Inference Capabilities to Edge Devices*. 2020. URL: https://aws.amazon.com/greengrass/ (visited on Sept. 8, 2020).

[162] Amazon Web Services. *AWS IoT - IoT Services for Industrial, Consumer, and Commercial Solutions*. 2020. URL: https://aws.amazon.com/de/iot/ (visited on Sept. 8, 2020).

[163] Amazon Web Services. *Azure IoT Hub - Managed Service to Enable Bi-Directional Communication between IoT Devices and Azure*. 2020. URL: https://azure.microsoft.com/en-us/services/iot-hub/ (visited on Sept. 8, 2020).

[164] Amazon Web Services. *Introducing the Amazon Time Sync Service*. 2017. URL: https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/ (visited on Oct. 21, 2020).

[165] Vinay Setty, Maarten van Steen, Roman Vitenberg, and Spyros Voulgaris. "PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-Based Pub/Sub". In: *Middleware 2012*. Springer, 2012, pp. 271–291. DOI: 10.1007/978-3-642-35170-9_14.

[166] Nursultan Shabykeev. "Scaling GeoBroker Clusters Based on Local Sharding". MA thesis. Technische Universität Berlin, 2020.

[167] Yogeshwer Sharma et al. "Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services". In: *12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2015, pp. 351–366.

[168] Shashank Shekhar, Ajay Chhokra, Hongyang Sun, Aniruddha Gokhale, Abhishek Dubey, Xenofon Koutsoukos, and Gabor Karsai. "URMILA: Dynamically Trading-off Fog and Edge Resources for Performance and Mobility-aware IoT Services". In: *Journal of Systems Architecture* 107 (2020), pp. 101710–101729. DOI: 10.1016/j.sysarc.2020.101710.

[169] Jonathan de Carvalho Silva, Antonio M. Alberti, Petar Solic, and Andre L. L. Aquino. "LoRaWAN - A Low Power WAN Protocol for Internet of Things: a Review and Opportunities". In: *2017 2nd International Multidisciplinary Conference on Computer and Energy Science*. IEEE, 2017, pp. 1–6.

[170] Swaminathan Sivasubramanian, Michal Szymaniak, and Guillaume Pierre. "Replication for Web Hosting Systems". In: *ACM Computing Surveys* 36.3 (2004), pp. 291–334. DOI: 10.1145/1035570.1035573.

[171] Olena Skarlat, Vasileios Karagiannis, Thomas Rausch, Kevin Bachmann, and Stefan Schulte. "A Framework for Optimization, Service Placement, and Runtime Operation in the Fog". In: *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing*. IEEE, 2018, pp. 164–173. DOI: 10.1109/UCC.2018.00025.

[172] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. "Optimized IoT Service Placement in the Fog". In: *Service Oriented Computing and Applications* 11.4 (2017), pp. 427–443. DOI: 10.1007/s11761-017-0219-8.

[173] Anders Skovsgaard and Christian S. Jensen. "Top-k Point of Interest Retrieval Using Standard Indexes". In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL '14*. ACM, 2014, pp. 173–182. DOI: 10.1145/2666310.2666399.

[174] SmartBear software. *Swagger - API Development for Everyone*. 2021. URL: https://swagger.io/ (visited on Feb. 26, 2021).

[175] Pareto Software. *World Cities Database*. 2019. URL: https://simplemaps.com/data/world-cities (visited on Apr. 11, 2019).

[176] Florian Stanek. "Efficient Message Routing in Geo-Distributed Pub/Sub-Systems". MA thesis. Technische Universität Berlin, 2019.

[177] Yunlei Sun, Xiuquan Qiao, Bo Cheng, and Junliang Chen. "A Low-Delay, Lightweight Publish/Subscribe Architecture for Delay-Sensitive IOT Services". In: *2013 IEEE 20th International Conference on Web Services*. IEEE, 2013, pp. 179–186. DOI: 10.1109/ICWS.2013.33.

[178] Yuuichi Teranishi, Ryohei Banno, and Toyokazu Akiyama. "Scalable and Locality-Aware Distributed Topic-based Pub/Sub Messaging for IoT". In: *2015 IEEE Global Communications Conference*. IEEE, 2015, pp. 1–7. DOI: 10.1109/GLOCOM.2015.7417305.

[179] The Linux Foundation. *Prometheus - From Metrics to Insight*. 2021. URL: https://prometheus.io (visited on Feb. 26, 2021).

[180] Ariel Tseitlin. "The Antifragile Organization". In: *Communications of the ACM* 56.8 (2013), pp. 40–44. DOI: 10.1145/2492007.2492022.

[181] Tsuyoshi Hombashi. *tcconfig - A tc Command Wrapper*. 2021. URL: https://tcconfig.readthedocs.io/en/latest/ (visited on Feb. 26, 2021).

[182] Prateeksha Varshney and Yogesh Simmhan. "Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions". In: *2017 IEEE 1st International Conference on Fog and Edge Computing*. IEEE, 2017, pp. 115–124. DOI: 10.1109/ICFEC.2017.20.

[183] Akanksha Verma, Department of Computer Science, Amity University, Gurgaon, India, Amita Khatana, Department of Computer Science, Amity University, Gurgaon, India, Sarika Chaudhary, and Department of Computer Science, Amity University, Gurgaon, India. "A Comparative Study of Black Box Testing and White Box Testing". In: *International Journal of Computer Sciences and Engineering* 5.12 (2017), pp. 301–304. DOI: 10.26438/ijcse/v5i12.301304.

[184] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. "AP-Tree: Efficiently Support Location-aware Publish/Subscribe". In: *The VLDB Journal* 24.6 (2015), pp. 823–848. DOI: 10.1007/s00778-015-0403-4.

[185] Philip Wette, Martin Draxler, and Arne Schwabe. "MaxiNet: Distributed Emulation of Software-defined Networks". In: *2014 IFIP Networking Conference*. IEEE, 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857078.

[186] Mario Winter, Mohsen Ekssir-Monfared, Harry M. Sneed, Richard Seidl, and Lars Borner. *Der Integrationstest*. First Edition. Hanser, 2013.

[187] Zheng Xu, Hui Zhang, Vijayan Sugumaran, Kim-Kwang Raymond Choo, Lin Mei, and Yiwei Zhu. "Participatory Sensing-based Semantic and Spatial Analysis of Urban Emergency Events using Mobile Social Media". In: *EURASIP Journal on Wireless Communications and Networking* 2016.1 (2016), pp. 1–9. DOI: 10.1186/s13638-016-0553-0.

[188] Yi Sun, Yubin Zhao, and Jochen Schiller. "An Indoor Positioning System based on Inertial Sensors in Smartphone". In: *2015 IEEE Wireless Communications and Networking Conference*. IEEE, 2015, pp. 2221–2226. DOI: 10.1109/WCNC.2015.7127812.

[189] Boyang Yu and Jianping Pan. "Location-aware Associated Data Placement for Geo-Distributed Data-Intensive Applications". In: *2015 IEEE Conference on Computer Communications*. IEEE, 2015, pp. 603–611. DOI: 10.1109/INFOCOM.2015.7218428.

[190] Andreas Zeidler and Ludger Fiege. "Mobility support with REBECA". In: *23rd International Conference on Distributed Computing Systems Workshops*. IEEE, 2003, pp. 354–360. DOI: 10.1109/ICDCSW.2003.1203579.

[191] Yukun Zeng, Mengyuan Chao, and Radu Stoleru. "EmuEdge: A Hybrid Emulator for Reproducible and Realistic Edge Computing Experiments". In: *2019 IEEE International Conference on Fog Computing*. IEEE, 2019, pp. 153–164. DOI: 10.1109/ICFC.2019.00027.

[192] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Nikhil Goyal, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. "The Cloud is Not Enough: Saving IoT from the Cloud". In: *7th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, 2015, pp. 1–7.

[193] Ye Zhao, Kyungbaek Kim, and Nalini Venkatasubramanian. "DYNATOPS: A Dynamic Topic-based Publish/Subscribe Architecture". In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems - DEBS '13*. ACM, 2013, pp. 75–86. DOI: 10.1145/2488222.2489273.

[194] Yu Zheng, Xing Xie, and Wei-Ying Ma. "GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory". In: *IEEE Data Engineering Bulletin* 33.2 (2010), pp. 32–39.

List of Figures

List of Tables

List of Algorithms

List of Listings